Approximation of functions

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Apr 22, 2015

PRELIMINARY VERSION

Contents

The finite element method is a powerful tool for solving differential equations. The method can easily deal with complex geometries and higher-order approximations of the solution. Figure 1 shows a two-dimensional domain with a non-trivial

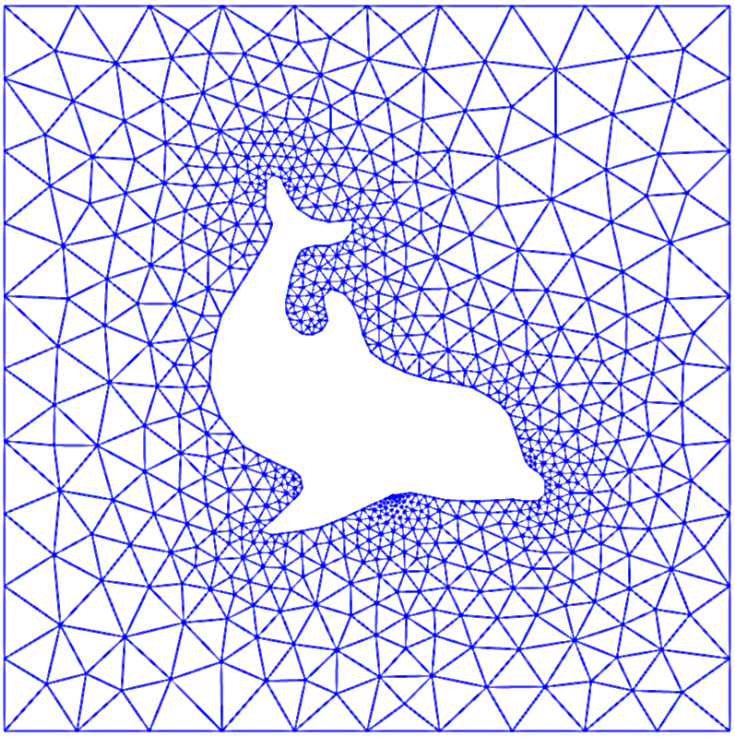

Figure 1: Domain for flow around a dolphin.

geometry. The idea is to divide the domain into triangles (elements) and seek a polynomial approximations to the unknown functions on each triangle. The method glues these piecewise approximations together to find a global solution. Linear and quadratic polynomials over the triangles are particularly popular.

Many successful numerical methods for differential equations, including the finite element method, aim at approximating the unknown function by a sum

$$u(x) = \sum_{i=0}^{N} c_i \psi_i(x), \tag{1}$$

where $\psi_i(x)$ are prescribed functions and $c_0, \ldots, c_N$ are unknown coefficients to be determined. Solution methods for differential equations utilizing (1) must have a *principle* for constructing $N+1$ equations to determine $c_0, \ldots, c_N$. Then there is a *machinery* regarding the actual constructions of the equations for $c_0, \ldots, c_N$, in a particular problem. Finally, there is a *solve* phase for computing the solution $c_0, \ldots, c_N$ of the $N+1$ equations.

Especially in the finite element method, the machinery for constructing the discrete equations to be implemented on a computer is quite comprehensive, with many mathematical and implementational details entering the scene at the same

time. From an ease-of-learning perspective it can therefore be wise to follow an idea of Larson and Bengzon [1] and introduce the computational machinery for a trivial equation: $u = f$. Solving this equation with f given and u on the form (1) means that we seek an approximation u to f. This approximation problem has the advantage of introducing most of the finite element toolbox, but with postponing demanding topics related to differential equations (e.g., integration by parts, boundary conditions, and coordinate mappings). This is the reason why we shall first become familiar with finite element *approximation* before addressing finite element methods for differential equations.

First, we refresh some linear algebra concepts about approximating vectors in vector spaces. Second, we extend these concepts to approximating functions in function spaces, using the same principles and the same notation. We present examples on approximating functions by global basis functions with support throughout the entire domain. Third, we introduce the finite element type of local basis functions and explain the computational algorithms for working with such functions. Three types of approximation principles are covered: 1) the least squares method, 2) the L_2 projection or Galerkin method, and 3) interpolation or collocation.

1 Approximation of vectors

We shall start with introducing two fundamental methods for determining the coefficients c_i in (1) and illustrate the methods on approximation of vectors, because vectors in vector spaces give a more intuitive understanding than starting directly with approximation of functions in function spaces. The extension from vectors to functions will be trivial as soon as the fundamental ideas are understood.

The first method of approximation is called the *least squares method* and consists in finding c_i such that the difference $u - f$, measured in some norm, is minimized. That is, we aim at finding the best approximation u to f (in some norm). The second method is not as intuitive: we find u such that the error $u - f$ is orthogonal to the space where we seek u. This is known as *projection*, or we may also call it a *Galerkin method*. When approximating vectors and functions, the two methods are equivalent, but this is no longer the case when applying the principles to differential equations.

1.1 Approximation of planar vectors

Suppose we have given a vector $\boldsymbol{f} = (3, 5)$ in the xy plane and that we want to approximate this vector by a vector aligned in the direction of the vector (a, b). Figure 2 depicts the situation.

We introduce the vector space V spanned by the vector $\boldsymbol{\psi}_0 = (a, b)$:

$$V = \text{span}\,\{\boldsymbol{\psi}_0\}\,. \tag{2}$$

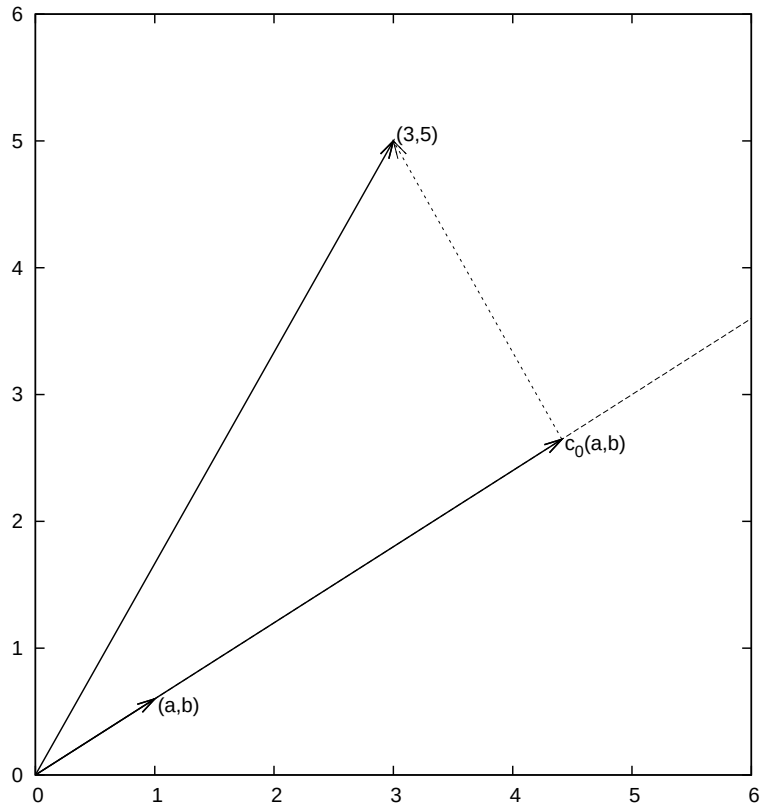

Figure 2: Approximation of a two-dimensional vector by a one-dimensional vector.

We say that $\boldsymbol{\psi}_0$ is a basis vector in the space V. Our aim is to find the vector $\boldsymbol{u} = c_0\boldsymbol{\psi}_0 \in V$ which best approximates the given vector $\boldsymbol{f} = (3, 5)$. A reasonable criterion for a best approximation could be to minimize the length of the difference between the approximate $\boldsymbol{u}$ and the given $\boldsymbol{f}$. The difference, or error $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$, has its length given by the *norm*

$$||\boldsymbol{e}|| = (\boldsymbol{e}, \boldsymbol{e})^{\frac{1}{2}},$$

where $(\boldsymbol{e}, \boldsymbol{e})$ is the *inner product* of $\boldsymbol{e}$ and itself. The inner product, also called *scalar product* or *dot product*, of two vectors $\boldsymbol{u} = (u_0, u_1)$ and $\boldsymbol{v} = (v_0, v_1)$ is defined as

$$(\boldsymbol{u}, \boldsymbol{v}) = u_0v_0 + u_1v_1\,. \tag{3}$$

Remark 1. We should point out that we use the notation $(\cdot, \cdot)$ for two different things: (a, b) for scalar quantities a and b means the vector starting in the origin

and ending in the point (a, b), while $(\boldsymbol{u}, \boldsymbol{v})$ with vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ means the inner product of these vectors. Since vectors are here written in boldface font there should be no confusion. We may add that the norm associated with this inner product is the usual Eucledian length of a vector.

Remark 2. It might be wise to refresh some basic linear algebra by consulting a textbook. Exercises 1 and 2 suggest specific tasks to regain familiarity with fundamental operations on inner product vector spaces.

The least squares method. We now want to find c_0 such that it minimizes $||\boldsymbol{e}||$. The algebra is simplified if we minimize the square of the norm, $||\boldsymbol{e}||^2 = (\boldsymbol{e}, \boldsymbol{e})$, instead of the norm itself. Define the function

$$E(c_0) = (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f} - c_0\boldsymbol{\psi}_0, \boldsymbol{f} - c_0\boldsymbol{\psi}_0)\,. \tag{4}$$

We can rewrite the expressions of the right-hand side in a more convenient form for further work:

$$E(c_0) = (\boldsymbol{f}, \boldsymbol{f}) - 2c_0(\boldsymbol{f}, \boldsymbol{\psi}_0) + c_0^2(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0)\,. \tag{5}$$

The rewrite results from using the following fundamental rules for inner product spaces:

$$(\alpha\boldsymbol{u}, \boldsymbol{v}) = \alpha(\boldsymbol{u}, \boldsymbol{v}), \quad \alpha \in \mathbb{R}, \tag{6}$$

$$(\boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w}) = (\boldsymbol{u}, \boldsymbol{w}) + (\boldsymbol{v}, \boldsymbol{w}), \tag{7}$$

$$(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{v}, \boldsymbol{u})\,. \tag{8}$$

Minimizing $E(c_0)$ implies finding c_0 such that

$$\frac{\partial E}{\partial c_0} = 0\,.$$

Differentiating (5) with respect to c_0 gives

$$\frac{\partial E}{\partial c_0} = -2(\boldsymbol{f}, \boldsymbol{\psi}_0) + 2c_0(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0)\,. \tag{9}$$

Setting the above expression equal to zero and solving for c_0 gives

$$c_0 = \frac{(\boldsymbol{f}, \boldsymbol{\psi}_0)}{(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0)}, \tag{10}$$

which in the present case with $\boldsymbol{\psi}_0 = (a, b)$ results in

$$c_0 = \frac{3a + 5b}{a^2 + b^2}\,. \tag{11}$$

For later, it is worth mentioning that setting the key equation (9) to zero can be rewritten as

$$(\boldsymbol{f} - c0\boldsymbol{\psi}_0, \boldsymbol{\psi}_0) = 0,$$

or

$$(\boldsymbol{e}, \boldsymbol{\psi}_0) = 0\,. \tag{12}$$

The projection method. We shall now show that minimizing $||\boldsymbol{e}||^2$ implies that $\boldsymbol{e}$ is orthogonal to *any* vector $\boldsymbol{v}$ in the space V. This result is visually quite clear from Figure 2 (think of other vectors along the line (a, b): all of them will lead to a larger distance between the approximation and $\boldsymbol{f}$). To see this result mathematically, we express any $\boldsymbol{v} \in V$ as $\boldsymbol{v} = s\boldsymbol{\psi}_0$ for any scalar parameter s, recall that two vectors are orthogonal when their inner product vanishes, and calculate the inner product

$$\begin{aligned}
(\boldsymbol{e}, s\boldsymbol{\psi}_0) &= (\boldsymbol{f} - c_0\boldsymbol{\psi}_0, s\boldsymbol{\psi}_0) \\
&= (\boldsymbol{f}, s\boldsymbol{\psi}_0) - (c_0\boldsymbol{\psi}_0, s\boldsymbol{\psi}_0) \\
&= s(\boldsymbol{f}, \boldsymbol{\psi}_0) - sc_0(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0) \\
&= s(\boldsymbol{f}, \boldsymbol{\psi}_0) - s\frac{(\boldsymbol{f}, \boldsymbol{\psi}_0)}{(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0)}(\boldsymbol{\psi}_0, \boldsymbol{\psi}_0) \\
&= s\left((\boldsymbol{f}, \boldsymbol{\psi}_0) - (\boldsymbol{f}, \boldsymbol{\psi}_0)\right) \\
&= 0\,.
\end{aligned}$$

Therefore, instead of minimizing the square of the norm, we could demand that $\boldsymbol{e}$ is orthogonal to any vector in V. This method is known as *projection*, because it is the same as projecting the vector onto the subspace. (The approach can also be referred to as a Galerkin method as explained at the end of Section 1.2.)

Mathematically the projection method is stated by the equation

$$(\boldsymbol{e}, \boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V\,. \tag{13}$$

An arbitrary $\boldsymbol{v} \in V$ can be expressed as $s\boldsymbol{\psi}_0$, $s \in \mathbb{R}$, and therefore (13) implies

$$(\boldsymbol{e}, s\boldsymbol{\psi}_0) = s(\boldsymbol{e}, \boldsymbol{\psi}_0) = 0,$$

which means that the error must be orthogonal to the basis vector in the space V:

$$(\boldsymbol{e}, \boldsymbol{\psi}_0) = 0 \quad \text{or} \quad (\boldsymbol{f} - c_0\boldsymbol{\psi}_0, \boldsymbol{\psi}_0) = 0\,.$$

The latter equation gives (10) and it also arose from least squares computations in (12).

1.2 Approximation of general vectors

Let us generalize the vector approximation from the previous section to vectors in spaces with arbitrary dimension. Given some vector $\boldsymbol{f}$, we want to find the best approximation to this vector in the space

$$V = \text{span}\,\{\boldsymbol{\psi}_0, \ldots, \boldsymbol{\psi}_N\}\,.$$

We assume that the *basis vectors* $\boldsymbol{\psi}_0, \ldots, \boldsymbol{\psi}_N$ are linearly independent so that none of them are redundant and the space has dimension $N+1$. Any vector $\boldsymbol{u} \in V$ can be written as a linear combination of the basis vectors,

$$\boldsymbol{u} = \sum_{j=0}^{N} c_j \boldsymbol{\psi}_j,$$

where $c_j \in \mathbb{R}$ are scalar coefficients to be determined.

The least squares method. Now we want to find $c_0, \ldots, c_N$, such that $\boldsymbol{u}$ is the best approximation to $\boldsymbol{f}$ in the sense that the distance (error) $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$ is minimized. Again, we define the squared distance as a function of the free parameters $c_0, \ldots, c_N$,

$$\begin{aligned} E(c_0, \ldots, c_N) &= (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f} - \sum_j c_j \boldsymbol{\psi}_j, \boldsymbol{f} - \sum_j c_j \boldsymbol{\psi}_j) \\ &= (\boldsymbol{f}, \boldsymbol{f}) - 2\sum_{j=0}^{N} c_j (\boldsymbol{f}, \boldsymbol{\psi}_j) + \sum_{p=0}^{N}\sum_{q=0}^{N} c_p c_q (\boldsymbol{\psi}_p, \boldsymbol{\psi}_q)\,. \end{aligned} \tag{14}$$

Minimizing this E with respect to the independent variables $c_0, \ldots, c_N$ is obtained by requiring

$$\frac{\partial E}{\partial c_i} = 0, \quad i = 0, \ldots, N\,.$$

The second term in (14) is differentiated as follows:

$$\frac{\partial}{\partial c_i} \sum_{j=0}^{N} c_j (\boldsymbol{f}, \boldsymbol{\psi}_j) = (\boldsymbol{f}, \boldsymbol{\psi}_i), \tag{15}$$

since the expression to be differentiated is a sum and only one term, $c_i(\boldsymbol{f}, \boldsymbol{\psi}_i)$, contains c_i and this term is linear in c_i. To understand this differentiation in detail, write out the sum specifically for, e.g, $N = 3$ and $i = 1$.

The last term in (14) is more tedious to differentiate. We start with

$$\frac{\partial}{\partial c_i} c_p c_q = \begin{cases} 0, & \text{if } p \neq i \text{ and } q \neq i, \\ c_q, & \text{if } p = i \text{ and } q \neq i, \\ c_p, & \text{if } p \neq i \text{ and } q = i, \\ 2c_i, & \text{if } p = q = i, \end{cases} \tag{16}$$

Then

$$\frac{\partial}{\partial c_i}\sum_{p=0}^{N}\sum_{q=0}^{N} c_p c_q(\boldsymbol{\psi}_p,\boldsymbol{\psi}_q) = \sum_{p=0,p\neq i}^{N} c_p(\boldsymbol{\psi}_p,\boldsymbol{\psi}_i) + \sum_{q=0,q\neq i}^{N} c_q(\boldsymbol{\psi}_q,\boldsymbol{\psi}_i) + 2c_i(\boldsymbol{\psi}_i,\boldsymbol{\psi}_i)\,.$$

The last term can be included in the other two sums, resulting in

$$\frac{\partial}{\partial c_i}\sum_{p=0}^{N}\sum_{q=0}^{N} c_p c_q(\boldsymbol{\psi}_p,\boldsymbol{\psi}_q) = 2\sum_{j=0}^{N} c_i(\boldsymbol{\psi}_j,\boldsymbol{\psi}_i)\,. \tag{17}$$

It then follows that setting

$$\frac{\partial E}{\partial c_i} = 0, \quad i = 0,\ldots,N,$$

leads to a linear system for $c_0,\ldots,c_N$:

$$\sum_{j=0}^{N} A_{i,j}c_j = b_i, \quad i = 0,\ldots,N, \tag{18}$$

where

$$A_{i,j} = (\boldsymbol{\psi}_i,\boldsymbol{\psi}_j), \tag{19}$$
$$b_i = (\boldsymbol{\psi}_i,\boldsymbol{f})\,. \tag{20}$$

We have changed the order of the two vectors in the inner product according to (1.1):

$$A_{i,j} = (\boldsymbol{\psi}_j,\boldsymbol{\psi}_i) = (\boldsymbol{\psi}_i,\boldsymbol{\psi}_j),$$

simply because the sequence i-j looks more aesthetic.

The Galerkin or projection method. In analogy with the "one-dimensional" example in Section 1.1, it holds also here in the general case that minimizing the distance (error) $\boldsymbol{e}$ is equivalent to demanding that $\boldsymbol{e}$ is orthogonal to all $\boldsymbol{v}\in V$:

$$(\boldsymbol{e},\boldsymbol{v}) = 0, \quad \forall \boldsymbol{v}\in V\,. \tag{21}$$

Since any $\boldsymbol{v}\in V$ can be written as $\boldsymbol{v} = \sum_{i=0}^{N} c_i\boldsymbol{\psi}_i$, the statement (21) is equivalent to saying that

$$(\boldsymbol{e},\sum_{i=0}^{N} c_i\boldsymbol{\psi}_i) = 0,$$

for any choice of coefficients $c_0,\ldots,c_N$. The latter equation can be rewritten as

$$\sum_{i=0}^{N} c_i(\boldsymbol{e}, \boldsymbol{\psi}_i) = 0\,.$$

If this is to hold for arbitrary values of $c_0, \ldots, c_N$ we must require that each term in the sum vanishes,

$$(\boldsymbol{e}, \boldsymbol{\psi}_i) = 0, \quad i = 0, \ldots, N\,. \tag{22}$$

These $N+1$ equations result in the same linear system as (18):

$$(\boldsymbol{f} - \sum_{j=0}^{N} c_j \boldsymbol{\psi}_j, \boldsymbol{\psi}_i) = (\boldsymbol{f}, \boldsymbol{\psi}_i) - \sum_{j \in \mathcal{I}_s} (\boldsymbol{\psi}_i, \boldsymbol{\psi}_j) c_j = 0,$$

and hence

$$\sum_{j=0}^{N} (\boldsymbol{\psi}_i, \boldsymbol{\psi}_j) c_j = (\boldsymbol{f}, \boldsymbol{\psi}_i), \quad i = 0, \ldots, N\,.$$

So, instead of differentiating the $E(c_0, \ldots, c_N)$ function, we could simply use (21) as the principle for determining $c_0, \ldots, c_N$, resulting in the $N+1$ equations (22).

The names *least squares method* or *least squares approximation* are natural since the calculations consists of minimizing $||\boldsymbol{e}||^2$, and $||\boldsymbol{e}||^2$ is a sum of squares of differences between the components in $\boldsymbol{f}$ and $\boldsymbol{u}$. We find $\boldsymbol{u}$ such that this sum of squares is minimized.

The principle (21), or the equivalent form (22), is known as *projection*. Almost the same mathematical idea was used by the Russian mathematician Boris Galerkin[1] to solve differential equations, resulting in what is widely known as *Galerkin's method*.

2 Approximation of functions

Let V be a function space spanned by a set of *basis functions* $\psi_0, \ldots, \psi_N$,

$$V = \text{span}\,\{\psi_0, \ldots, \psi_N\},$$

such that any function $u \in V$ can be written as a linear combination of the basis functions:

$$u = \sum_{j \in \mathcal{I}_s} c_j \psi_j\,. \tag{23}$$

The index set $\mathcal{I}_s$ is defined as $\mathcal{I}_s = \{0, \ldots, N\}$ and is used both for compact notation and for flexibility in the numbering of elements in sequences.

[1] `http://en.wikipedia.org/wiki/Boris_Galerkin`

For now, in this introduction, we shall look at functions of a single variable x: $u = u(x)$, $\psi_i = \psi_i(x)$, $i \in \mathcal{I}_s$. Later, we will almost trivially extend the mathematical details to functions of two- or three-dimensional physical spaces. The approximation (23) is typically used to discretize a problem in space. Other methods, most notably finite differences, are common for time discretization, although the form (23) can be used in time as well.

2.1 The least squares method

Given a function $f(x)$, how can we determine its best approximation $u(x) \in V$? A natural starting point is to apply the same reasoning as we did for vectors in Section 1.2. That is, we minimize the distance between u and f. However, this requires a norm for measuring distances, and a norm is most conveniently defined through an inner product. Viewing a function as a vector of infinitely many point values, one for each value of x, the inner product could intuitively be defined as the usual summation of pairwise components, with summation replaced by integration:

$$(f, g) = \int f(x)g(x)\ \mathrm{d}x\,.$$

To fix the integration domain, we let $f(x)$ and $\psi_i(x)$ be defined for a domain $\Omega \subset \mathbb{R}$. The inner product of two functions $f(x)$ and $g(x)$ is then

$$(f, g) = \int_\Omega f(x)g(x)\ \mathrm{d}x\,. \tag{24}$$

The distance between f and any function $u \in V$ is simply $f - u$, and the squared norm of this distance is

$$E = (f(x) - \sum_{j\in\mathcal{I}_s} c_j\psi_j(x), f(x) - \sum_{j\in\mathcal{I}_s} c_j\psi_j(x))\,. \tag{25}$$

Note the analogy with (14): the given function f plays the role of the given vector $\boldsymbol{f}$, and the basis function ψ_i plays the role of the basis vector $\boldsymbol{\psi}_i$. We can rewrite (25), through similar steps as used for the result (14), leading to

$$E(c_i, \ldots, c_N) = (f, f) - 2\sum_{j\in\mathcal{I}_s} c_j(f, \psi_i) + \sum_{p\in\mathcal{I}_s}\sum_{q\in\mathcal{I}_s} c_p c_q(\psi_p, \psi_q)\,. \tag{26}$$

Minimizing this function of $N+1$ scalar variables $\{c_i\}_{i\in\mathcal{I}_s}$, requires differentiation with respect to c_i, for all $i \in \mathcal{I}_s$. The resulting equations are very similar to those we had in the vector case, and we hence end up with a linear system of the form (18), with basically the same expressions:

$$A_{i,j} = (\psi_i, \psi_j), \tag{27}$$
$$b_i = (f, \psi_i)\,. \tag{28}$$

2.2 The projection (or Galerkin) method

As in Section 1.2, the minimization of (e, e) is equivalent to

$$(e, v) = 0, \quad \forall v \in V\,. \tag{29}$$

This is known as a projection of a function f onto the subspace V. We may also call it a Galerkin method for approximating functions. Using the same reasoning as in (21)-(22), it follows that (29) is equivalent to

$$(e, \psi_i) = 0, \quad i \in \mathcal{I}_s\,. \tag{30}$$

Inserting $e = f - u$ in this equation and ordering terms, as in the multi-dimensional vector case, we end up with a linear system with a coefficient matrix (27) and right-hand side vector (28).

Whether we work with vectors in the plane, general vectors, or functions in function spaces, the least squares principle and the projection or Galerkin method are equivalent.

2.3 Example: linear approximation

Let us apply the theory in the previous section to a simple problem: given a parabola $f(x) = 10(x-1)^2 - 1$ for $x \in \Omega = [1, 2]$, find the best approximation $u(x)$ in the space of all linear functions:

$$V = \text{span}\,\{1, x\}\,.$$

With our notation, $\psi_0(x) = 1$, $\psi_1(x) = x$, and $N = 1$. We seek

$$u = c_0\psi_0(x) + c_1\psi_1(x) = c_0 + c_1 x,$$

where c_0 and c_1 are found by solving a 2×2 the linear system. The coefficient matrix has elements

$$A_{0,0} = (\psi_0, \psi_0) = \int_1^2 1 \cdot 1 \,\mathrm{d}x = 1, \tag{31}$$

$$A_{0,1} = (\psi_0, \psi_1) = \int_1^2 1 \cdot x \,\mathrm{d}x = 3/2, \tag{32}$$

$$A_{1,0} = A_{0,1} = 3/2, \tag{33}$$

$$A_{1,1} = (\psi_1, \psi_1) = \int_1^2 x \cdot x \,\mathrm{d}x = 7/3\,. \tag{34}$$

The corresponding right-hand side is

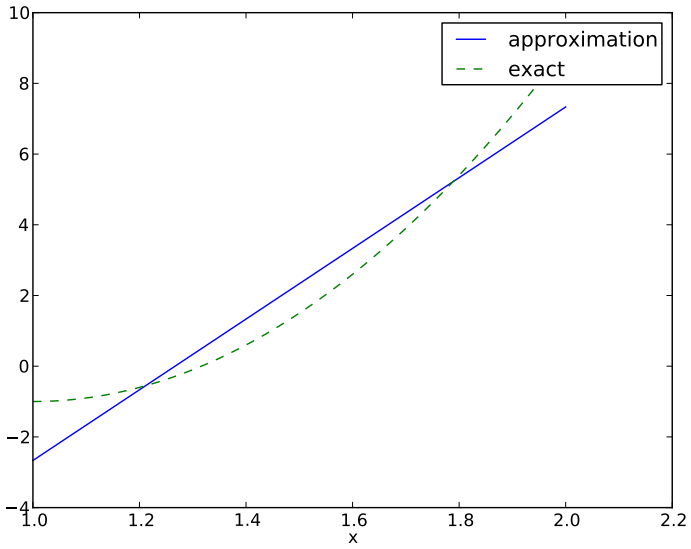

Figure 3: Best approximation of a parabola by a straight line.

$$b_1 = (f, \psi_0) = \int_1^2 (10(x-1)^2 - 1) \cdot 1 \, \mathrm{d}x = 7/3, \tag{35}$$

$$b_2 = (f, \psi_1) = \int_1^2 (10(x-1)^2 - 1) \cdot x \, \mathrm{d}x = 13/3\,. \tag{36}$$

Solving the linear system results in

$$c_0 = -38/3, \quad c_1 = 10, \tag{37}$$

and consequently

$$u(x) = 10x - \frac{38}{3}\,. \tag{38}$$

Figure 3 displays the parabola and its best approximation in the space of all linear functions.

2.4 Implementation of the least squares method

Symbolic integration. The linear system can be computed either symbolically or numerically (a numerical integration rule is needed in the latter case). Here is a function for symbolic computation of the linear system, where $f(x)$ is

given as a `sympy` expression `f` involving the symbol `x`, `psi` is a list of expressions for $\{\psi_i\}_{i\in\mathcal{I}_s}$, and `Omega` is a 2-tuple/list holding the limits of the domain Ω:

```
import sympy as sp

def least_squares(f, psi, Omega):
    N = len(psi) - 1
    A = sp.zeros((N+1, N+1))
    b = sp.zeros((N+1, 1))
    x = sp.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            A[i,j] = sp.integrate(psi[i]*psi[j],
                                  (x, Omega[0], Omega[1]))
            A[j,i] = A[i,j]
        b[i,0] = sp.integrate(psi[i]*f, (x, Omega[0], Omega[1]))
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i]
    return u, c
```

Observe that we exploit the symmetry of the coefficient matrix: only the upper triangular part is computed. Symbolic integration in `sympy` is often time consuming, and (roughly) halving the work has noticeable effect on the waiting time for the function to finish execution.

Fallback on numerical integration. Obviously, `sympy` mail fail to successfully integrate $\int_\Omega \psi_i\psi_j dx$ and especially $\int_\Omega f\psi_i dx$ symbolically. Therefore, we should extend the `least_squares` function such that it falls back on numerical integration if the symbolic integration is unsuccessful. In the latter case, the returned value from `sympy`'s `integrate` function is an object of type `Integral`. We can test on this type and utilize the `mpmath` module in `sympy` to perform numerical integration of high precision. Even when `sympy` manages to integrate symbolically, it can take an undesirable long time. We therefore include an argument `symbolic` that governs whether or not to try symbolic integration. Here is the complete code of the improved version of function `least_squares`:

```
def least_squares(f, psi, Omega, symbolic=True):
    N = len(psi) - 1
    A = sp.zeros((N+1, N+1))
    b = sp.zeros((N+1, 1))
    x = sp.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = psi[i]*psi[j]
            if symbolic:
                I = sp.integrate(integrand, (x, Omega[0], Omega[1]))
            if not symbolic or isinstance(I, sp.Integral):
                # Could not integrate symbolically,
                # fall back on numerical integration
                integrand = sp.lambdify([x], integrand)
                I = sp.mpmath.quad(integrand, [Omega[0], Omega[1]])
            A[i,j] = A[j,i] = I
```

```
        integrand = psi[i]*f
        if symbolic:
            I = sp.integrate(integrand, (x, Omega[0], Omega[1]))
        if not symbolic or isinstance(I, sp.Integral):
            integrand = sp.lambdify([x], integrand)
            I = sp.mpmath.quad(integrand, [Omega[0], Omega[1]])
        b[i,0] = I
    c = A.LUsolve(b)  # symbolic solve
    # c is a sympy Matrix object, numbers are in c[i,0]
    u = sum(c[i,0]*psi[i] for i in range(len(psi)))
    return u, [c[i,0] for i in range(len(c))]
```

The function is found in the file `approx1D.py`.

Plotting the approximation. Comparing the given $f(x)$ and the approximate $u(x)$ visually is done by the following function, which with the aid of `sympy`'s `lambdify` tool converts a `sympy` expression to a Python function for numerical computations:

```
def comparison_plot(f, u, Omega, filename='tmp.pdf'):
    x = sp.Symbol('x')
    f = sp.lambdify([x], f, modules="numpy")
    u = sp.lambdify([x], u, modules="numpy")
    resolution = 401  # no of points in plot
    xcoor  = linspace(Omega[0], Omega[1], resolution)
    exact  = f(xcoor)
    approx = u(xcoor)
    plot(xcoor, approx)
    hold('on')
    plot(xcoor, exact)
    legend(['approximation', 'exact'])
    savefig(filename)
```

The `modules='numpy'` argument to `lambdify` is important if there are mathematical functions, such as `sin` or `exp` in the symbolic expressions in `f` or `u`, and these mathematical functions are to be used with vector arguments, like `xcoor` above.

Both the `least_squares` and `comparison_plot` are found and coded in the file `approx1D.py`[2]. The forthcoming examples on their use appear in `ex_approx1D.py`.

2.5 Perfect approximation

Let us use the code above to recompute the problem from Section 2.3 where we want to approximate a parabola. What happens if we add an element x^2 to the basis and test what the best approximation is if V is the space of all parabolic functions? The answer is quickly found by running

[2] `http://tinyurl.com/nm5587k/fem/approx1D.py`

```
>>> from approx1D import *
>>> x = sp.Symbol('x')
>>> f = 10*(x-1)**2-1
>>> u, c = least_squares(f=f, psi=[1, x, x**2], Omega=[1, 2])
>>> print u
10*x**2 - 20*x + 9
>>> print sp.expand(f)
10*x**2 - 20*x + 9
```

Now, what if we use $\psi_i(x) = x^i$ for $i = 0, 1, \ldots, N = 40$? The output from `least_squares` gives $c_i = 0$ for $i > 2$, which means that the method finds the perfect approximation.

In fact, we have a general result that if $f \in V$, the least squares and projection/Galerkin methods compute the exact solution $u = f$. The proof is straightforward: if $f \in V$, f can be expanded in terms of the basis functions, $f = \sum_{j\in\mathcal{I}_s} d_j\psi_j$, for some coefficients $\{d_i\}_{i\in\mathcal{I}_s}$, and the right-hand side then has entries

$$b_i = (f, \psi_i) = \sum_{j\in\mathcal{I}_s} d_j(\psi_j, \psi_i) = \sum_{j\in\mathcal{I}_s} d_j A_{i,j}\,.$$

The linear system $\sum_j A_{i,j}c_j = b_i$, $i \in \mathcal{I}_s$, is then

$$\sum_{j\in\mathcal{I}_s} c_j A_{i,j} = \sum_{j\in\mathcal{I}_s} d_j A_{i,j}, \quad i \in \mathcal{I}_s,$$

which implies that $c_i = d_i$ for $i \in \mathcal{I}_s$.

2.6 Ill-conditioning

The computational example in Section 2.5 applies the `least_squares` function which invokes symbolic methods to calculate and solve the linear system. The correct solution $c_0 = 9, c_1 = -20, c_2 = 10, c_i = 0$ for $i \geq 3$ is perfectly recovered.

Suppose we convert the matrix and right-hand side to floating-point arrays and then solve the system using finite-precision arithmetics, which is what one will (almost) always do in real life. This time we get astonishing results! Up to about $N = 7$ we get a solution that is reasonably close to the exact one. Increasing N shows that seriously wrong coefficients are computed. Below is a table showing the solution of the linear system arising from approximating a parabola by functions on the form $u(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{10}x^{10}$. Analytically, we know that $c_j = 0$ for $j > 2$, but numerically we may get $c_j \neq 0$ for $j > 2$.

exact	`sympy`	`numpy32`	`numpy64`
9	9.62	5.57	8.98
-20	-23.39	-7.65	-19.93
10	17.74	-4.50	9.96
0	-9.19	4.13	-0.26
0	5.25	2.99	0.72
0	0.18	-1.21	-0.93
0	-2.48	-0.41	0.73
0	1.81	-0.013	-0.36
0	-0.66	0.08	0.11
0	0.12	0.04	-0.02
0	-0.001	-0.02	0.002

The exact value of c_j, $j = 0, 1, \ldots, 10$, appears in the first column while the other columns correspond to results obtained by three different methods:

- Column 2: The matrix and vector are converted to the data structure `sympy.mpmath.fp.matrix` and the `sympy.mpmath.fp.lu_solve` function is used to solve the system.
- Column 3: The matrix and vector are converted to `numpy` arrays with data type `numpy.float32` (single precision floating-point number) and solved by the `numpy.linalg.solve` function.
- Column 4: As column 3, but the data type is `numpy.float64` (double precision floating-point number).

We see from the numbers in the table that double precision performs much better than single precision. Nevertheless, when plotting all these solutions the curves cannot be visually distinguished (!). This means that the approximations look perfect, despite the partially very wrong values of the coefficients.

Increasing N to 12 makes the numerical solver in `numpy` abort with the message: "matrix is numerically singular". A matrix has to be non-singular to be invertible, which is a requirement when solving a linear system. Already when the matrix is close to singular, it is *ill-conditioned*, which here implies that the numerical solution algorithms are sensitive to round-off errors and may produce (very) inaccurate results.

The reason why the coefficient matrix is nearly singular and ill-conditioned is that our basis functions $\psi_i(x) = x^i$ are nearly linearly dependent for large i. That is, x^i and x^{i+1} are very close for i not very small. This phenomenon is illustrated in Figure 4. There are 15 lines in this figure, but only half of them are visually distinguishable. Almost linearly dependent basis functions give rise to an ill-conditioned and almost singular matrix. This fact can be illustrated by computing the determinant, which is indeed very close to zero (recall that a zero determinant implies a singular and non-invertible matrix): 10^{-65} for $N = 10$ and 10^{-92} for $N = 12$. Already for $N = 28$ the numerical determinant computation returns a plain zero.

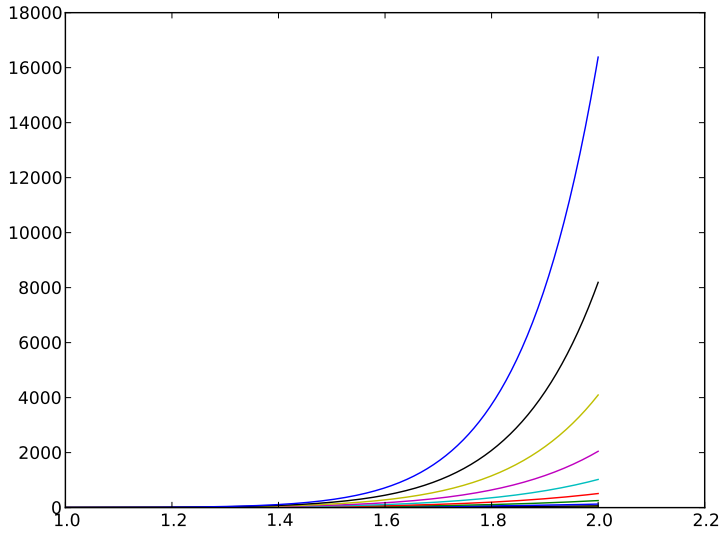

Figure 4: The 15 first basis functions x^i, $i = 0, \ldots, 14$.

On the other hand, the double precision `numpy` solver do run for $N = 100$, resulting in answers that are not significantly worse than those in the table above, and large powers are associated with small coefficients (e.g., $c_j < 10^{-2}$ for $10 \leq j \leq 20$ and $c < 10^{-5}$ for $j > 20$). Even for $N = 100$ the approximation still lies on top of the exact curve in a plot (!).

The conclusion is that visual inspection of the quality of the approximation may not uncover fundamental numerical problems with the computations. However, numerical analysts have studied approximations and ill-conditioning for decades, and it is well known that the basis $\{1, x, x^2, x^3, \ldots, \}$ is a bad basis. The best basis from a matrix conditioning point of view is to have orthogonal functions such that $(\psi_i, \psi_j) = 0$ for $i \neq j$. There are many known sets of orthogonal polynomials and other functions. The functions used in the finite element methods are almost orthogonal, and this property helps to avoid problems with solving matrix systems. Almost orthogonal is helpful, but not enough when it comes to partial differential equations, and ill-conditioning of the coefficient matrix is a theme when solving large-scale matrix systems arising from finite element discretizations.

2.7 Fourier series

A set of sine functions is widely used for approximating functions (the sines are also orthogonal as explained more in Section 2.6). Let us take

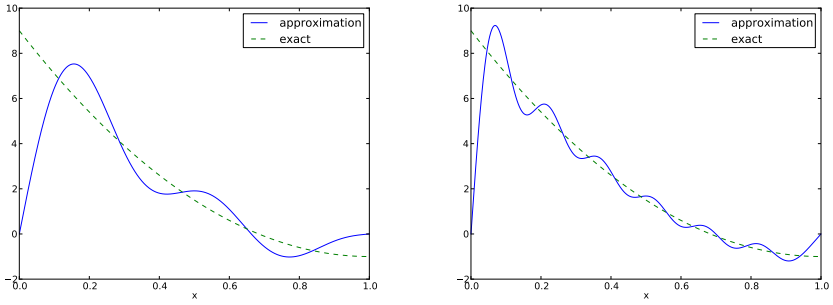

Figure 5: Best approximation of a parabola by a sum of 3 (left) and 11 (right) sine functions.

$$V = \text{span}\left\{\sin \pi x, \sin 2\pi x, \ldots, \sin(N+1)\pi x\right\}.$$

That is,

$$\psi_i(x) = \sin((i+1)\pi x), \quad i \in \mathcal{I}_s\,.$$

An approximation to the $f(x)$ function from Section 2.3 can then be computed by the `least_squares` function from Section 2.4:

```
N = 3
from sympy import sin, pi
x = sp.Symbol('x')
psi = [sin(pi*(i+1)*x) for i in range(N+1)]
f = 10*(x-1)**2 - 1
Omega = [0, 1]
u, c = least_squares(f, psi, Omega)
comparison_plot(f, u, Omega)
```

Figure 5 (left) shows the oscillatory approximation of $\sum_{j=0}^{N} c_j \sin((j+1)\pi x)$ when $N = 3$. Changing N to 11 improves the approximation considerably, see Figure 5 (right).

There is an error $f(0) - u(0) = 9$ at $x = 0$ in Figure 5 regardless of how large N is, because all $\psi_i(0) = 0$ and hence $u(0) = 0$. We may help the approximation to be correct at $x = 0$ by seeking

$$u(x) = f(0) + \sum_{j\in\mathcal{I}_s} c_j\psi_j(x)\,. \tag{39}$$

However, this adjustment introduces a new problem at $x = 1$ since we now get an error $f(1) - u(1) = f(1) - 0 = -1$ at this point. A more clever adjustment is to replace the $f(0)$ term by a term that is $f(0)$ at $x = 0$ and $f(1)$ at $x = 1$. A simple linear combination $f(0)(1-x) + xf(1)$ does the job:

$$u(x) = f(0)(1-x) + xf(1) + \sum_{j\in\mathcal{I}_s} c_j\psi_j(x)\,. \tag{40}$$

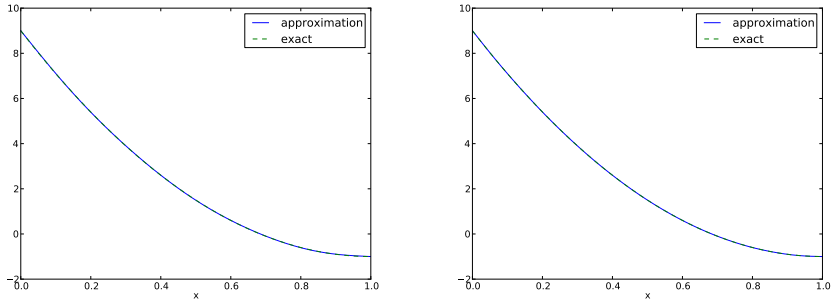

Figure 6: Best approximation of a parabola by a sum of 3 (left) and 11 (right) sine functions with a boundary term.

This adjustment of u alters the linear system slightly. In the general case, we set

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x),$$

and the linear system becomes

$$\sum_{j \in \mathcal{I}_s} (\psi_i, \psi_j) c_j = (f - B, \psi_i), \quad i \in \mathcal{I}_s \,.$$

The calculations can still utilize the `least_squares` or `least_squares_orth` functions, but solve for $u - b$:

```
f0 = 0;  f1 = -1
B = f0*(1-x) + x*f1
u_sum, c = least_squares_orth(f-b, psi, Omega)
u = B + u_sum
```

Figure 6 shows the result of the technique for ensuring right boundary values. Even 3 sines can now adjust the $f(0)(1-x)+xf(1)$ term such that u approximates the parabola really well, at least visually.

2.8 Orthogonal basis functions

The choice of sine functions $\psi_i(x) = \sin((i+1)\pi x)$ has a great computational advantage: on $\Omega = [0,1]$ these basis functions are *orthogonal*, implying that $A_{i,j} = 0$ if $i \neq j$. This result is realized by trying

```
integrate(sin(j*pi*x)*sin(k*pi*x), x, 0, 1)
```

in WolframAlpha[3] (avoid `i` in the integrand as this symbol means the imaginary unit $\sqrt{-1}$). Also by asking WolframAlpha about $\int_0^1 \sin^2(j\pi x)\,\mathrm{d}x$, we find it to

[3] `http://wolframalpha.com`

equal 1/2. With a diagonal matrix we can easily solve for the coefficients by hand:

$$c_i = 2\int_0^1 f(x)\sin((i+1)\pi x)\,\mathrm{d}x, \quad i \in \mathcal{I}_s, \tag{41}$$

which is nothing but the classical formula for the coefficients of the Fourier sine series of $f(x)$ on $[0,1]$. In fact, when V contains the basic functions used in a Fourier series expansion, the approximation method derived in Section 2 results in the classical Fourier series for $f(x)$ (see Exercise 8 for details).

With orthogonal basis functions we can make the `least_squares` function (much) more efficient since we know that the matrix is diagonal and only the diagonal elements need to be computed:

```
def least_squares_orth(f, psi, Omega):
    N = len(psi) - 1
    A = [0]*(N+1)
    b = [0]*(N+1)
    x = sp.Symbol('x')
    for i in range(N+1):
        A[i] = sp.integrate(psi[i]**2, (x, Omega[0], Omega[1]))
        b[i] = sp.integrate(psi[i]*f,  (x, Omega[0], Omega[1]))
    c = [b[i]/A[i] for i in range(len(b))]
    u = 0
    for i in range(len(psi)):
        u += c[i]*psi[i]
    return u, c
```

As mentioned in Section 2.4, symbolic integration may fail or take very long time. It is therefore natural to extend the implementation above with a version where we can choose between symbolic and numerical integration and fall back on the latter if the former fails:

```
def least_squares_orth(f, psi, Omega, symbolic=True):
    N = len(psi) - 1
    A = [0]*(N+1)       # plain list to hold symbolic expressions
    b = [0]*(N+1)
    x = sp.Symbol('x')
    for i in range(N+1):
        # Diagonal matrix term
        A[i] = sp.integrate(psi[i]**2, (x, Omega[0], Omega[1]))

        # Right-hand side term
        integrand = psi[i]*f
        if symbolic:
            I = sp.integrate(integrand,  (x, Omega[0], Omega[1]))
        if not symbolic or isinstance(I, sp.Integral):
            print 'numerical integration of', integrand
            integrand = sp.lambdify([x], integrand)
            I = sp.mpmath.quad(integrand, [Omega[0], Omega[1]])
        b[i] = I
    c = [b[i]/A[i] for i in range(len(b))]
    u = 0
    u = sum(c[i,0]*psi[i] for i in range(len(psi)))
    return u, c
```

This function is found in the file `approx1D.py`. Observe that we here assume that $\int_\Omega \varphi_i^2 dx$ can always be symbolically computed, which is not an unreasonable assumption when the basis functions are orthogonal, but there is no guarantee, so an improved version of the function above would implement numerical integration also for the `A[i,i]` term.

2.9 Numerical computations

Sometimes the basis functions ψ_i and/or the function f have a nature that makes symbolic integration CPU-time consuming or impossible. Even though we implemented a fallback on numerical integration of $\int f\varphi_i dx$ considerable time might be required by `sympy` in the attempt to integrate symbolically. Therefore, it will be handy to have function for fast *numerical integration and numerical solution of the linear system*. Below is such a method. It requires Python functions `f(x)` and `psi(x,i)` for $f(x)$ and $\psi_i(x)$ as input. The output is a mesh function with values `u` on the mesh with points in the array `x`. Three numerical integration methods are offered: `scipy.integrate.quad` (precision set to 10^{-8}), `sympy.mpmath.quad` (high precision), and a Trapezoidal rule based on the points in `x`.

```
def least_squares_numerical(f, psi, N, x,
                            integration_method='scipy',
                            orthogonal_basis=False):
    import scipy.integrate
    A = np.zeros((N+1, N+1))
    b = np.zeros(N+1)
    Omega = [x[0], x[-1]]
    dx = x[1] - x[0]

    for i in range(N+1):
        j_limit = i+1 if orthogonal_basis else N+1
        for j in range(i, j_limit):
            print '(%d,%d)' % (i, j)
            if integration_method == 'scipy':
                A_ij = scipy.integrate.quad(
                    lambda x: psi(x,i)*psi(x,j),
                    Omega[0], Omega[1], epsabs=1E-9, epsrel=1E-9)[0]
            elif integration_method == 'sympy':
                A_ij = sp.mpmath.quad(
                    lambda x: psi(x,i)*psi(x,j),
                    [Omega[0], Omega[1]])
            else:
                values = psi(x,i)*psi(x,j)
                A_ij = trapezoidal(values, dx)
            A[i,j] = A[j,i] = A_ij

        if integration_method == 'scipy':
            b_i = scipy.integrate.quad(
                lambda x: f(x)*psi(x,i), Omega[0], Omega[1],
                epsabs=1E-9, epsrel=1E-9)[0]
        elif integration_method == 'sympy':
            b_i = sp.mpmath.quad(
                lambda x: f(x)*psi(x,i), [Omega[0], Omega[1]])
        else:
```

```
            values = f(x)*psi(x,i)
            b_i = trapezoidal(values, dx)
        b[i] = b_i

    c = b/np.diag(A) if orthogonal_basis else np.linalg.solve(A, b)
    u = sum(c[i]*psi(x, i) for i in range(N+1))
    return u, c

def trapezoidal(values, dx):
    """Integrate values by the Trapezoidal rule (mesh size dx)."""
    return dx*(np.sum(values) - 0.5*values[0] - 0.5*values[-1])
```

Here is an example on calling the function:

```
from numpy import linspace, tanh, pi

def psi(x, i):
    return sin((i+1)*x)

x = linspace(0, 2*pi, 501)
N = 20
u, c = least_squares_numerical(lambda x: tanh(x-pi), psi, N, x,
                               orthogonal_basis=True)
```

2.10 The interpolation (or collocation) method

The principle of minimizing the distance between u and f is an intuitive way of computing a best approximation $u \in V$ to f. However, there are other approaches as well. One is to demand that $u(x_i) = f(x_i)$ at some selected points x_i, $i \in \mathcal{I}_s$:

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x_i) = f(x_i), \quad i \in \mathcal{I}_s \,. \tag{42}$$

This criterion also gives a linear system with $N+1$ unknown coefficients $\{c_i\}_{i \in \mathcal{I}_s}$:

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s, \tag{43}$$

with

$$A_{i,j} = \psi_j(x_i), \tag{44}$$

$$b_i = f(x_i) \,. \tag{45}$$

This time the coefficient matrix is not symmetric because $\psi_j(x_i) \neq \psi_i(x_j)$ in general. The method is often referred to as an *interpolation method* since some point values of f are given ($f(x_i)$) and we fit a continuous function u that goes through the $f(x_i)$ points. In this case the x_i points are called *interpolation points*. When the same approach is used to approximate differential equations, one usually applies the name *collocation method* and x_i are known as *collocation points*.

Given f as a `sympy` symbolic expression `f`, $\{\psi_i\}_{i\in\mathcal{I}_s}$ as a list `psi`, and a set of points $\{x_i\}_{i\in\mathcal{I}_s}$ as a list or array `points`, the following Python function sets up and solves the matrix system for the coefficients $\{c_i\}_{i\in\mathcal{I}_s}$:

```
def interpolation(f, psi, points):
    N = len(psi) - 1
    A = sp.zeros((N+1, N+1))
    b = sp.zeros((N+1, 1))
    x = sp.Symbol('x')
    # Turn psi and f into Python functions
    psi = [sp.lambdify([x], psi[i]) for i in range(N+1)]
    f = sp.lambdify([x], f)
    for i in range(N+1):
        for j in range(N+1):
            A[i,j] = psi[j](points[i])
        b[i,0] = f(points[i])
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i](x)
    return u
```

The `interpolation` function is a part of the `approx1D` module.

We found it convenient in the above function to turn the expressions `f` and `psi` into ordinary Python functions of `x`, which can be called with `float` values in the list `points` when building the matrix and the right-hand side. The alternative is to use the `subs` method to substitute the `x` variable in an expression by an element from the `points` list. The following session illustrates both approaches in a simple setting:

```
>>> from sympy import *
>>> x = Symbol('x')
>>> e = x**2              # symbolic expression involving x
>>> p = 0.5               # a value of x
>>> v = e.subs(x, p)      # evaluate e for x=p
>>> v
0.250000000000000
>>> type(v)
sympy.core.numbers.Float
>>> e = lambdify([x], e)  # make Python function of e
>>> type(e)
>>> function
>>> v = e(p)              # evaluate e(x) for x=p
>>> v
0.25
>>> type(v)
float
```

A nice feature of the interpolation or collocation method is that it avoids computing integrals. However, one has to decide on the location of the x_i points. A simple, yet common choice, is to distribute them uniformly throughout Ω.

Example. Let us illustrate the interpolation or collocation method by approximating our parabola $f(x) = 10(x-1)^2 - 1$ by a linear function on $\Omega = [1, 2]$, using two collocation points $x_0 = 1 + 1/3$ and $x_1 = 1 + 2/3$:

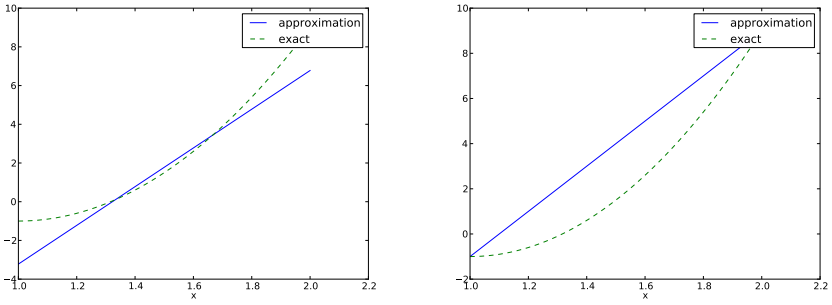

Figure 7: Approximation of a parabola by linear functions computed by two interpolation points: 4/3 and 5/3 (left) versus 1 and 2 (right).

```
f = 10*(x-1)**2 - 1
psi = [1, x]
Omega = [1, 2]
points = [1 + sp.Rational(1,3), 1 + sp.Rational(2,3)]
u = interpolation(f, psi, points)
comparison_plot(f, u, Omega)
```

The resulting linear system becomes

$$\begin{pmatrix} 1 & 4/3 \\ 1 & 5/3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1/9 \\ 31/9 \end{pmatrix}$$

with solution $c_0 = -119/9$ and $c_1 = 10$. Figure 7 (left) shows the resulting approximation $u = -119/9 + 10x$. We can easily test other interpolation points, say $x_0 = 1$ and $x_1 = 2$. This changes the line quite significantly, see Figure 7 (right).

2.11 Lagrange polynomials

In Section 2.7 we explain the advantage with having a diagonal matrix: formulas for the coefficients $\{c_i\}_{i\in\mathcal{I}_s}$ can then be derived by hand. For an interpolation/-collocation method a diagonal matrix implies that $\psi_j(x_i) = 0$ if $i \neq j$. One set of basis functions $\psi_i(x)$ with this property is the *Lagrange interpolating polynomials*, or just *Lagrange polynomials*. (Although the functions are named after Lagrange, they were first discovered by Waring in 1779, rediscovered by Euler in 1783, and published by Lagrange in 1795.) The Lagrange polynomials have the form

$$\psi_i(x) = \prod_{j=0, j\neq i}^{N} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_N}{x_i - x_N}, \quad (46)$$

for $i \in \mathcal{I}_s$. We see from (46) that all the ψ_i functions are polynomials of degree N which have the property

$$\psi_i(x_s) = \delta_{is}, \quad \delta_{is} = \begin{cases} 1, & i = s, \\ 0, & i \neq s, \end{cases} \tag{47}$$

when x_s is an interpolation/collocation point. Here we have used the *Kronecker delta* symbol δ_{is}. This property implies that $A_{i,j} = 0$ for $i \neq j$ and $A_{i,j} = 1$ when $i = j$. The solution of the linear system is them simply

$$c_i = f(x_i), \quad i \in \mathcal{I}_s, \tag{48}$$

and

$$u(x) = \sum_{j \in \mathcal{I}_s} f(x_i)\psi_i(x)\,. \tag{49}$$

The following function computes the Lagrange interpolating polynomial $\psi_i(x)$, given the interpolation points $x_0, \ldots, x_N$ in the list or array `points`:

```
def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p
```

The next function computes a complete basis using equidistant points throughout Ω:

```
def Lagrange_polynomials_01(x, N):
    if isinstance(x, sp.Symbol):
        h = sp.Rational(1, N-1)
    else:
        h = 1.0/(N-1)
    points = [i*h for i in range(N)]
    psi = [Lagrange_polynomial(x, i, points) for i in range(N)]
    return psi, points
```

When `x` is an `sp.Symbol` object, we let the spacing between the interpolation points, `h`, be a `sympy` rational number for nice end results in the formulas for ψ_i. The other case, when `x` is a plain Python `float`, signifies numerical computing, and then we let `h` be a floating-point number. Observe that the `Lagrange_polynomial` function works equally well in the symbolic and numerical case - just think of `x` being an `sp.Symbol` object or a Python `float`. A little interactive session illustrates the difference between symbolic and numerical computing of the basis functions and points:

```
>>> import sympy as sp
>>> x = sp.Symbol('x')
>>> psi, points = Lagrange_polynomials_01(x, N=3)
>>> points
[0, 1/2, 1]
>>> psi
```

```
[(1 - x)*(1 - 2*x), 2*x*(2 - 2*x), -x*(1 - 2*x)]

>>> x = 0.5  # numerical computing
>>> psi, points = Lagrange_polynomials_01(x, N=3)
>>> points
[0.0, 0.5, 1.0]
>>> psi
[-0.0, 1.0, 0.0]
```

The Lagrange polynomials are very much used in finite element methods because of their property (47).

Approximation of a polynomial. The Galerkin or least squares method lead to an exact approximation if f lies in the space spanned by the basis functions. It could be interest to see how the interpolation method with Lagrange polynomials as basis is able to approximate a polynomial, e.g., a parabola. Running

```
for N in 2, 4, 5, 6, 8, 10, 12:
    f = x**2
    psi, points = Lagrange_polynomials_01(x, N)
    u = interpolation(f, psi, points)
```

shows the result that up to `N=4` we achieve an exact approximation, and then round-off errors start to grow, such that `N=15` leads to a 15-degree polynomial for u where the coefficients in front of x^r for $r > 2$ are of size 10^{-5} and smaller.

Successful example. Trying out the Lagrange polynomial basis for approximating $f(x) = \sin 2\pi x$ on $\Omega = [0, 1]$ with the least squares and the interpolation techniques can be done by

```
x = sp.Symbol('x')
f = sp.sin(2*sp.pi*x)
psi, points = Lagrange_polynomials_01(x, N)
Omega=[0, 1]
u, c = least_squares(f, psi, Omega)
comparison_plot(f, u, Omega)
u, c = interpolation(f, psi, points)
comparison_plot(f, u, Omega)
```

Figure 8 shows the results. There is little difference between the least squares and the interpolation technique. Increasing N gives visually better approximations.

Less successful example. The next example concerns interpolating $f(x) = |1 - 2x|$ on $\Omega = [0, 1]$ using Lagrange polynomials. Figure 9 shows a peculiar effect: the approximation starts to oscillate more and more as N grows. This numerical artifact is not surprising when looking at the individual Lagrange polynomials. Figure 10 shows two such polynomials, $\psi_2(x)$ and $\psi_7(x)$, both of degree 11 and computed from uniformly spaced points $x_{x_i} = i/11$, $i = 0, \ldots, 11$, marked with circles. We clearly see the property of Lagrange polynomials: $\psi_2(x_i) = 0$ and $\psi_7(x_i) = 0$ for all i, except $\psi_2(x_2) = 1$ and $\psi_7(x_7) = 1$. The

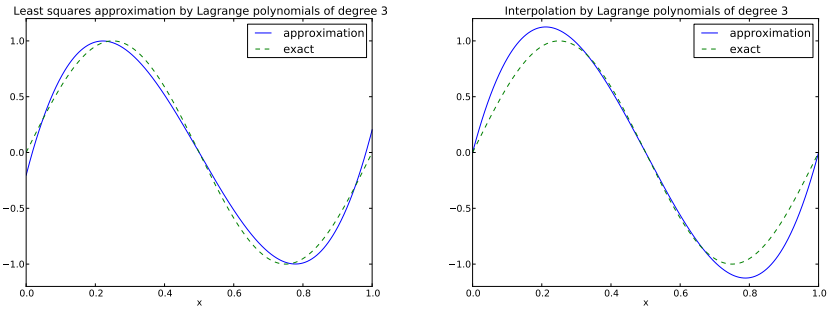

Figure 8: Approximation via least squares (left) and interpolation (right) of a sine function by Lagrange interpolating polynomials of degree 3.

most striking feature, however, is the significant oscillation near the boundary. The reason is easy to understand: since we force the functions to zero at so many points, a polynomial of high degree is forced to oscillate between the points. The phenomenon is named *Runge's phenomenon* and you can read a more detailed explanation on Wikipedia[4].

Remedy for strong oscillations. The oscillations can be reduced by a more clever choice of interpolation points, called the *Chebyshev nodes*:

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2i+1}{2(N+1)}pi\right), \quad i = 0\ldots, N, \tag{50}$$

on the interval $\Omega = [a, b]$. Here is a flexible version of the `Lagrange_polynomials_01` function above, valid for any interval $\Omega = [a, b]$ and with the possibility to generate both uniformly distributed points and Chebyshev nodes:

```
def Lagrange_polynomials(x, N, Omega, point_distribution='uniform'):
    if point_distribution == 'uniform':
        if isinstance(x, sp.Symbol):
            h = sp.Rational(Omega[1] - Omega[0], N)
        else:
            h = (Omega[1] - Omega[0])/float(N)
        points = [Omega[0] + i*h for i in range(N+1)]
    elif point_distribution == 'Chebyshev':
        points = Chebyshev_nodes(Omega[0], Omega[1], N)
    psi = [Lagrange_polynomial(x, i, points) for i in range(N+1)]
    return psi, points

def Chebyshev_nodes(a, b, N):
    from math import cos, pi
    return [0.5*(a+b) + 0.5*(b-a)*cos(float(2*i+1)/(2*N+1))*pi) \
            for i in range(N+1)]
```

All the functions computing Lagrange polynomials listed above are found in the module file `Lagrange.py`. Figure 11 shows the improvement of using Chebyshev

[4] `http://en.wikipedia.org/wiki/Runge%27s_phenomenon`

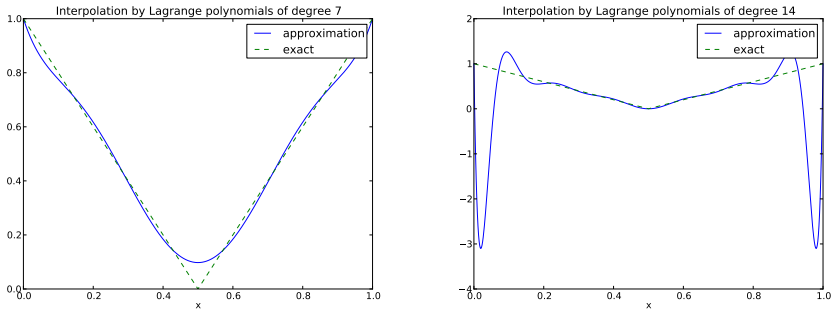

Figure 9: Interpolation of an absolute value function by Lagrange polynomials and uniformly distributed interpolation points: degree 7 (left) and 14 (right).

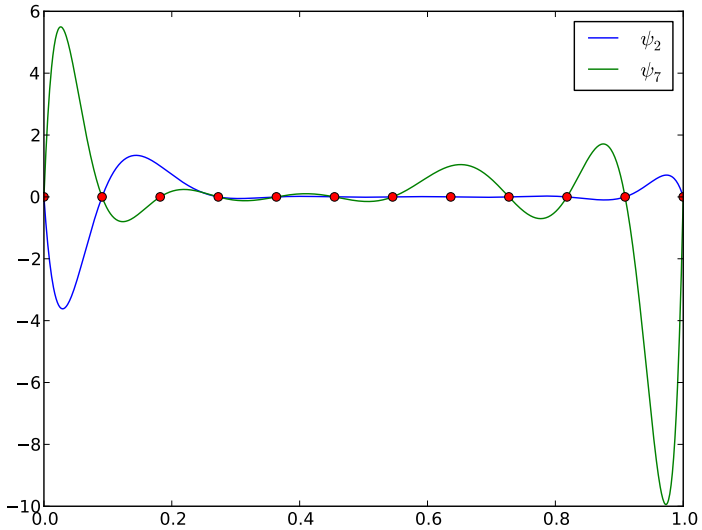

Figure 10: Illustration of the oscillatory behavior of two Lagrange polynomials based on 12 uniformly spaced points (marked by circles).

nodes (compared with Figure 9). The reason is that the corresponding Lagrange polynomials have much smaller oscillations as seen in Figure 12 (compare with Figure 10).

Another cure for undesired oscillation of higher-degree interpolating polynomials is to use lower-degree Lagrange polynomials on many small patches of the domain, which is the idea pursued in the finite element method. For instance, linear Lagrange polynomials on $[0, 1/2]$ and $[1/2, 1]$ would yield a perfect approximation to $f(x) = |1 - 2x|$ on $\Omega = [0, 1]$ since f is piecewise linear.

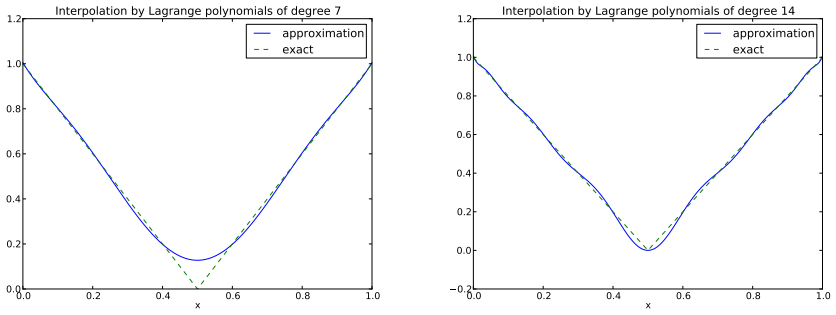

Figure 11: Interpolation of an absolute value function by Lagrange polynomials and Chebyshev nodes as interpolation points: degree 7 (left) and 14 (right).

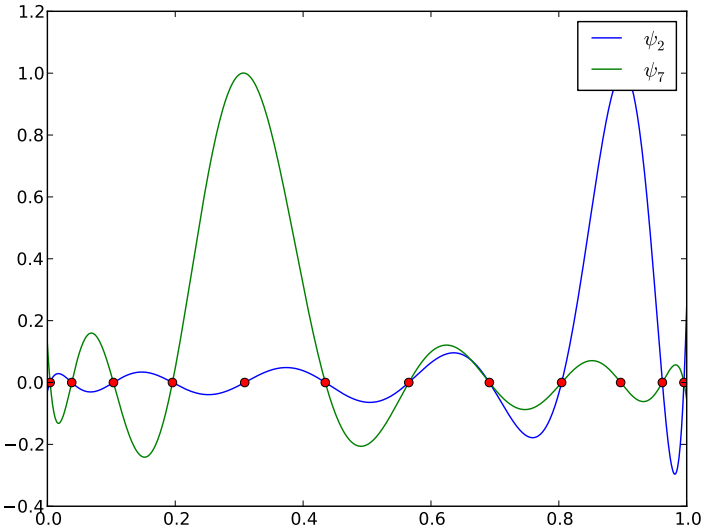

Figure 12: Illustration of the less oscillatory behavior of two Lagrange polynomials based on 12 Chebyshev points (marked by circles).

How does the least squares or projection methods work with Lagrange polynomials? We can just call the `least_squares` function, but `sympy` has problems integrating the $f(x) = |1 - 2x|$ function times a polynomial, so we need to fallback on numerical integration.

```
def least_squares(f, psi, Omega):
    N = len(psi) - 1
    A = sp.zeros((N+1, N+1))
    b = sp.zeros((N+1, 1))
```

```
    x = sp.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = psi[i]*psi[j]
            I = sp.integrate(integrand, (x, Omega[0], Omega[1]))
            if isinstance(I, sp.Integral):
                # Could not integrate symbolically, fallback
                # on numerical integration with mpmath.quad
                integrand = sp.lambdify([x], integrand)
                I = sp.mpmath.quad(integrand, [Omega[0], Omega[1]])
            A[i,j] = A[j,i] = I
        integrand = psi[i]*f
        I = sp.integrate(integrand, (x, Omega[0], Omega[1]))
        if isinstance(I, sp.Integral):
            integrand = sp.lambdify([x], integrand)
            I = sp.mpmath.quad(integrand, [Omega[0], Omega[1]])
        b[i,0] = I
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i]
    return u
```

3 Finite element basis functions

The specific basis functions exemplified in Section 2 are in general nonzero on the entire domain Ω, see Figure 13 for an example where we plot $\psi_0(x) = \sin\frac{1}{2}\pi x$ and $\psi_1(x) = \sin 2\pi x$ together with a possible sum $u(x) = 4\psi_0(x) - \frac{1}{2}\psi_1(x)$. We shall now turn the attention to basis functions that have *compact support*, meaning that they are nonzero on only a small portion of Ω. Moreover, we shall restrict the functions to be *piecewise polynomials*. This means that the domain is split into subdomains and the function is a polynomial on one or more subdomains, see Figure 14 for a sketch involving locally defined hat functions that make $u = \sum_j c_j\psi_j$ piecewise linear. At the boundaries between subdomains one normally forces continuity of the function only so that when connecting two polynomials from two subdomains, the derivative becomes discontinuous. These type of basis functions are fundamental in the finite element method.

We first introduce the concepts of elements and nodes in a simplistic fashion as often met in the literature. Later, we shall generalize the concept of an element, which is a necessary step to treat a wider class of approximations within the family of finite element methods. The generalization is also compatible with the concepts used in the FEniCS[5] finite element software.

3.1 Elements and nodes

Let us divide the interval Ω on which f and u are defined into N_e non-overlapping subintervals $\Omega^{(e)}$, $e = 0, \ldots, N_e - 1$:

[5] `http://fenicsproject.org`

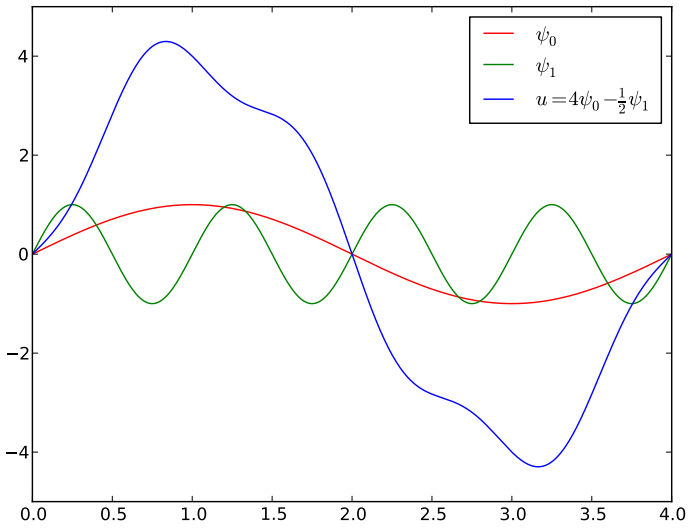

Figure 13: A function resulting from adding two sine basis functions.

$$\Omega = \Omega^{(0)} \cup \cdots \cup \Omega^{(N_e)}. \tag{51}$$

We shall for now refer to $\Omega^{(e)}$ as an *element*, having number e. On each element we introduce a set of points called *nodes*. For now we assume that the nodes are uniformly spaced throughout the element and that the boundary points of the elements are also nodes. The nodes are given numbers both within an element and in the global domain. These are referred to as *local* and *global* node numbers, respectively. Local nodes are numbered with an index $r = 0, \ldots, d$, while the N_n global nodes are numbered as $i = 0, \ldots, N_n - 1$. Figure 15 shows element boundaries with small vertical lines, nodes as small disks, element numbers in circles, and global node numbers under the nodes.

Nodes and elements uniquely define a *finite element mesh*, which is our discrete representation of the domain in the computations. A common special case is that of a *uniformly partitioned mesh* where each element has the same length and the distance between nodes is constant.

Example. On $\Omega = [0, 1]$ we may introduce two elements, $\Omega^{(0)} = [0, 0.4]$ and $\Omega^{(1)} = [0.4, 1]$. Furthermore, let us introduce three nodes per element, equally spaced within each element. Figure 16 shows the mesh with $N_e = 2$ elements and $N_n = 2N_e + 1 = 5$ nodes. The three nodes in element number 0 are $x_0 = 0$, $x_1 = 0.2$, and $x_2 = 0.4$. The local and global node numbers are here equal. In element number 1, we have the local nodes $x_0 = 0.4$, $x_1 = 0.7$, and $x_2 = 1$ and

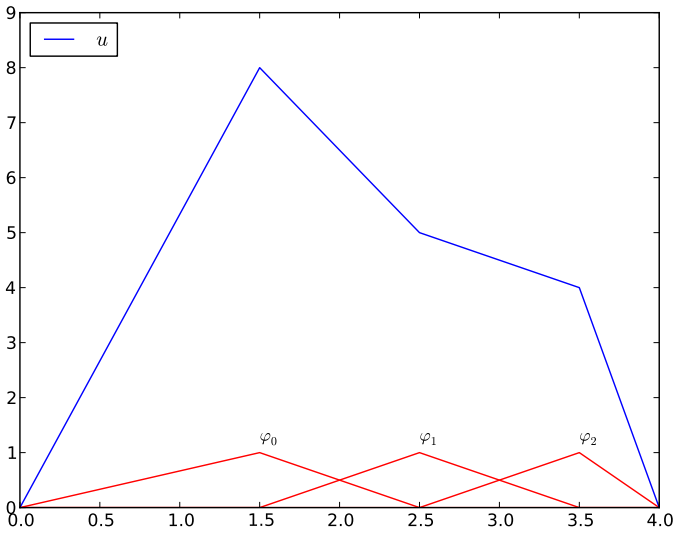

Figure 14: A function resulting from adding three local piecewise linear (hat) functions.

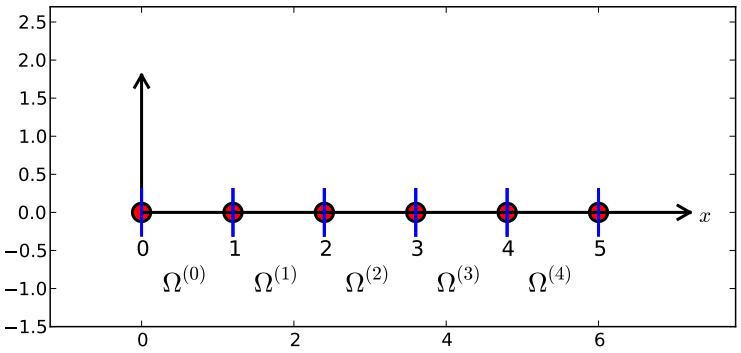

Figure 15: Finite element mesh with 5 elements and 6 nodes.

the corresponding global nodes $x_2 = 0.4$, $x_3 = 0.7$, and $x_4 = 1$. Note that the global node $x_2 = 0.4$ is shared by the two elements.

For the purpose of implementation, we introduce two lists or arrays: `nodes` for storing the coordinates of the nodes, with the global node numbers as indices, and `elements` for holding the global node numbers in each element, with the local node numbers as indices. The `nodes` and `elements` lists for the sample mesh above take the form

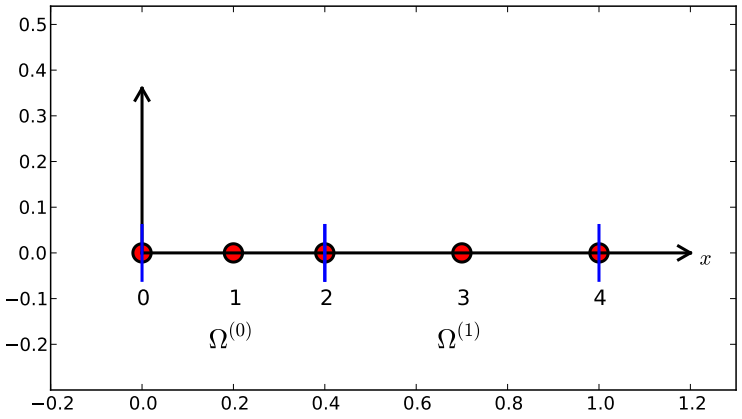

Figure 16: Finite element mesh with 2 elements and 5 nodes.

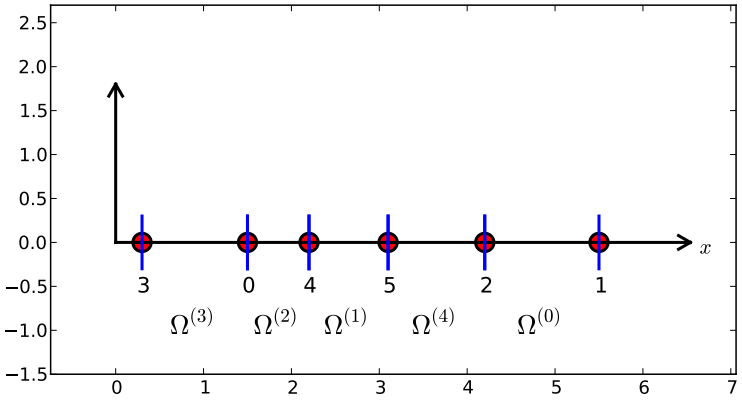

Figure 17: Example on irregular numbering of elements and nodes.

```
nodes = [0, 0.2, 0.4, 0.7, 1]
elements = [[0, 1, 2], [2, 3, 4]]
```

Looking up the coordinate of local node number 2 in element 1 is here done by `nodes[elements[1][2]]` (recall that nodes and elements start their numbering at 0).

The numbering of elements and nodes does not need to be regular. Figure 17 shows and example corresponding to

```
nodes = [1.5, 5.5, 4.2, 0.3, 2.2, 3.1]
elements = [[2, 1], [4, 5], [0, 4], [3, 0], [5, 2]]
```

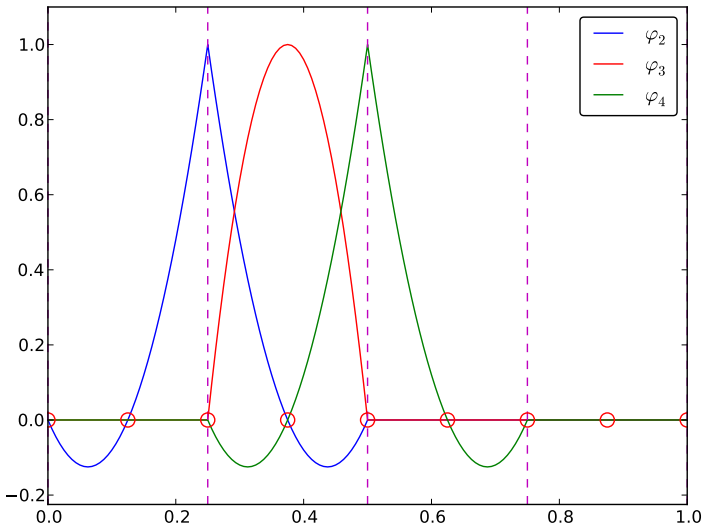

Figure 18: Illustration of the piecewise quadratic basis functions associated with nodes in element 1.

3.2 The basis functions

Construction principles. Finite element basis functions are in this text recognized by the notation $\varphi_i(x)$, where the index now in the beginning corresponds to a global node number. In the current approximation problem we shall simply take $\psi_i = \varphi_i$.

Let i be the global node number corresponding to local node r in element number e. The finite element basis functions φ_i are now defined as follows.

- If local node number r is not on the boundary of the element, take $\varphi_i(x)$ to be the Lagrange polynomial that is 1 at the local node number r and zero at all other nodes in the element. On all other elements, $\varphi_i = 0$.
- If local node number r is on the boundary of the element, let φ_i be made up of the Lagrange polynomial over element e that is 1 at node i, combined with the Lagrange polynomial over element $e+1$ that is also 1 at node i. On all other elements, $\varphi_i = 0$.

A visual impression of three such basis functions are given in Figure 18.

Properties of φ_i. The construction of basis functions according to the principles above lead to two important properties of $\varphi_i(x)$. First,

$$\varphi_i(x_j) = \delta_{ij}, \quad \delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j, \end{cases} \tag{52}$$

when x_j is a node in the mesh with global node number j. The result $\varphi_i(x_j) = \delta_{ij}$ arises because the Lagrange polynomials are constructed to have exactly this property. The property also implies a convenient interpretation of c_i as the value of u at node i. To show this, we expand u in the usual way as $\sum_j c_j\psi_j$ and choose $\psi_i = \varphi_i$:

$$u(x_i) = \sum_{j\in\mathcal{I}_s} c_j\psi_j(x_i) = \sum_{j\in\mathcal{I}_s} c_j\varphi_j(x_i) = c_i\varphi_i(x_i) = c_i\,.$$

Because of this interpretation, the coefficient c_i is by many named u_i or U_i.

Second, $\varphi_i(x)$ is mostly zero throughout the domain:

- $\varphi_i(x) \neq 0$ only on those elements that contain global node i,
- $\varphi_i(x)\varphi_j(x) \neq 0$ if and only if i and j are global node numbers in the same element.

Since $A_{i,j}$ is the integral of $\varphi_i\varphi_j$ it means that *most of the elements in the coefficient matrix will be zero*. We will come back to these properties and use them actively in computations to save memory and CPU time.

We let each element have $d+1$ nodes, resulting in local Lagrange polynomials of degree d. It is not a requirement to have the same d value in each element, but for now we will assume so.

3.3 Example on piecewise quadratic finite element functions

Figure 18 illustrates how piecewise quadratic basis functions can look like ($d = 2$). We work with the domain $\Omega = [0, 1]$ divided into four equal-sized elements, each having three nodes. The `nodes` and `elements` lists in this particular example become

```
nodes = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0]
elements = [[0, 1, 2], [2, 3, 4], [4, 5, 6], [6, 7, 8]]
```

Figure 19 sketches the mesh and the numbering. Nodes are marked with circles on the x axis and element boundaries are marked with vertical dashed lines in Figure 18.

Let us explain in detail how the basis functions are constructed according to the principles. Consider element number 1 in Figure 18, $\Omega^{(1)} = [0.25, 0.5]$, with local nodes 0, 1, and 2 corresponding to global nodes 2, 3, and 4. The coordinates of these nodes are 0.25, 0.375, and 0.5, respectively. We define three Lagrange polynomials on this element:

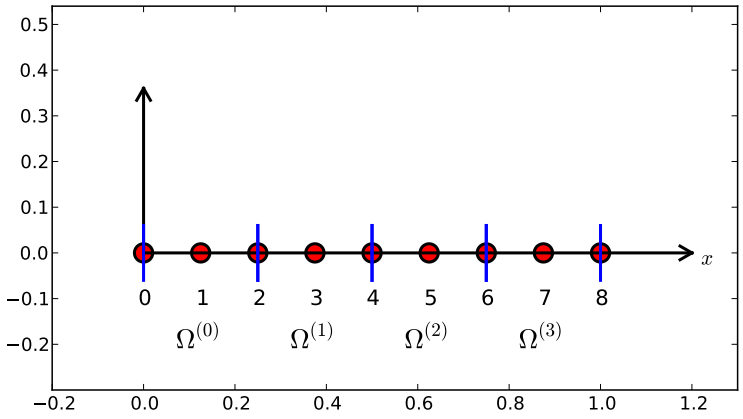

Figure 19: Sketch of mesh with 4 elements and 3 nodes per element.

1. The polynomial that is 1 at local node 1 ($x = 0.375$, global node 3) makes up the basis function $\varphi_3(x)$ over this element, with $\varphi_3(x) = 0$ outside the element.

2. The Lagrange polynomial that is 1 at local node 0 is the "right part" of the global basis function $\varphi_2(x)$. The "left part" of $\varphi_2(x)$ consists of a Lagrange polynomial associated with local node 2 in the neighboring element $\Omega^{(0)} = [0, 0.25]$.

3. Finally, the polynomial that is 1 at local node 2 (global node 4) is the "left part" of the global basis function $\varphi_4(x)$. The "right part" comes from the Lagrange polynomial that is 1 at local node 0 in the neighboring element $\Omega^{(2)} = [0.5, 0.75]$.

As mentioned earlier, any global basis function $\varphi_i(x)$ is zero on elements that do not contain the node with global node number i.

The other global functions associated with internal nodes, φ_1, φ_5, and φ_7, are all of the same shape as the drawn φ_3, while the global basis functions associated with shared nodes also have the same shape, provided the elements are of the same length.

3.4 Example on piecewise linear finite element functions

Figure 20 shows piecewise linear basis functions ($d = 1$). Also here we have four elements on $\Omega = [0, 1]$. Consider the element $\Omega^{(1)} = [0.25, 0.5]$. Now there are no internal nodes in the elements so that all basis functions are associated with nodes at the element boundaries and hence made up of two Lagrange polynomials from neighboring elements. For example, $\varphi_1(x)$ results from the Lagrange polynomial in element 0 that is 1 at local node 1 and 0 at local node 0, combined with the Lagrange polynomial in element 1 that is 1 at local node 0 and 0 at local node 1. The other basis functions are constructed similarly.

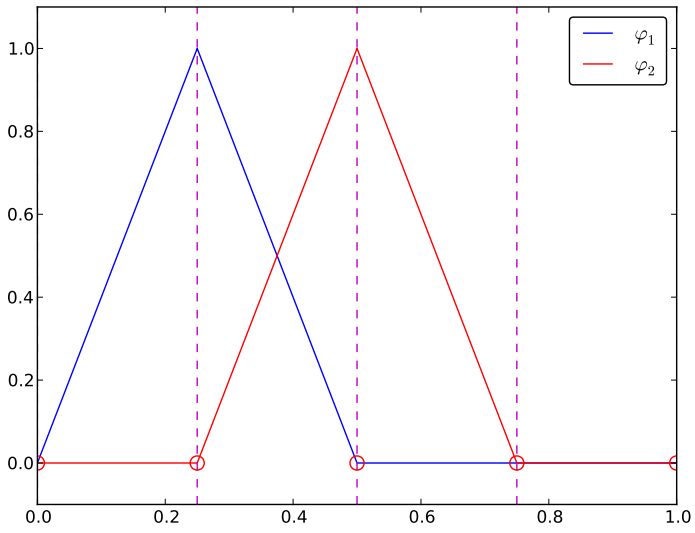

Figure 20: Illustration of the piecewise linear basis functions associated with nodes in element 1.

Explicit mathematical formulas are needed for $\varphi_i(x)$ in computations. In the piecewise linear case, one can show that

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/(x_i - x_{i-1}), & x_{i-1} \leq x < x_i, \\ 1 - (x - x_i)/(x_{i+1} - x_i), & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1}\,. \end{cases} \tag{53}$$

Here, x_j, $j = i-1, i, i+1$, denotes the coordinate of node j. For elements of equal length h the formulas can be simplified to

$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1}, \\ (x - x_{i-1})/h, & x_{i-1} \leq x < x_i, \\ 1 - (x - x_i)/h, & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1} \end{cases} \tag{54}$$

3.5 Example on piecewise cubic finite element basis functions

Piecewise cubic basis functions can be defined by introducing four nodes per element. Figure 21 shows examples on $\varphi_i(x)$, $i = 3, 4, 5, 6$, associated with element number 1. Note that φ_4 and φ_5 are nonzero on element number 1, while φ_3 and φ_6 are made up of Lagrange polynomials on two neighboring elements.

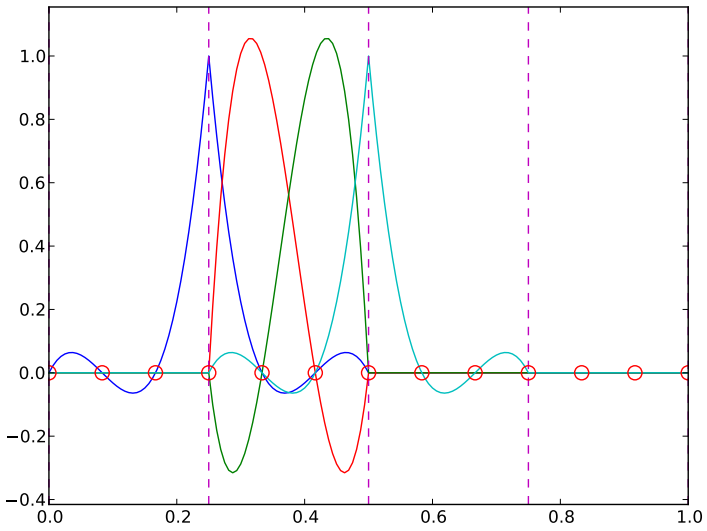

Figure 21: Illustration of the piecewise cubic basis functions associated with nodes in element 1.

We see that all the piecewise linear basis functions have the same "hat" shape. They are naturally referred to as *hat functions*, also called *chapeau functions*. The piecewise quadratic functions in Figure 18 are seen to be of two types. "Rounded hats" associated with internal nodes in the elements and some more "sombrero" shaped hats associated with element boundary nodes. Higher-order basis functions also have hat-like shapes, but the functions have pronounced oscillations in addition, as illustrated in Figure 21.

A common terminology is to speak about *linear elements* as elements with two local nodes associated with piecewise linear basis functions. Similarly, *quadratic elements* and *cubic elements* refer to piecewise quadratic or cubic functions over elements with three or four local nodes, respectively. Alternative names, frequently used later, are P1 elements for linear elements, P2 for quadratic elements, and so forth: Pd signifies degree d of the polynomial basis functions.

3.6 Calculating the linear system

The elements in the coefficient matrix and right-hand side are given by the formulas (27) and (28), but now the choice of ψ_i is φ_i. Consider P1 elements where $\varphi_i(x)$ piecewise linear. Nodes and elements numbered consecutively from left to right in a uniformly partitioned mesh imply the nodes

$$x_i = ih, \quad i = 0, \ldots, N_n - 1,$$

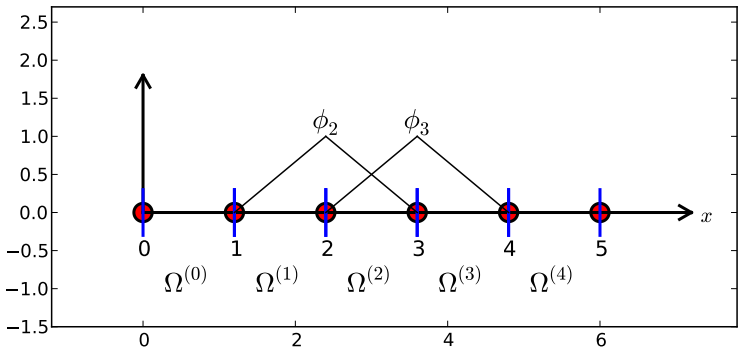

Figure 22: Illustration of the piecewise linear basis functions corresponding to global node 2 and 3.

and the elements

$$\Omega^{(i)} = [x_i, x_{i+1}] = [ih, (i+1)h], \quad i = 0, \ldots, N_e - 1\,. \tag{55}$$

We have in this case N_e elements and $N_n = N_e + 1$ nodes. The parameter N denotes the number of unknowns in the expansion for u, and with the P1 elements, $N = N_n - 1$. The domain is $\Omega = [x_0, x_N]$. The formula for $\varphi_i(x)$ is given by (54) and a graphical illustration is provided in Figures 20 and 23. First we clearly see from the figures the very important property $\varphi_i(x)\varphi_j(x) \neq 0$ if and only if $j = i - 1$, $j = i$, or $j = i + 1$, or alternatively expressed, if and only if i and j are nodes in the same element. Otherwise, φ_i and φ_j are too distant to have an overlap and consequently their product vanishes.

Calculating a specific matrix entry. Let us calculate the specific matrix entry $A_{2,3} = \int_\Omega \varphi_2\varphi_3\,\mathrm{d}x$. Figure 22 shows how φ_2 and φ_3 look like. We realize from this figure that the product $\varphi_2\varphi_3 \neq 0$ only over element 2, which contains node 2 and 3. The particular formulas for $\varphi_2(x)$ and $\varphi_3(x)$ on $[x_2, x_3]$ are found from (54). The function φ_3 has positive slope over $[x_2, x_3]$ and corresponds to the interval $[x_{i-1}, x_i]$ in (54). With $i = 3$ we get

$$\varphi_3(x) = (x - x_2)/h,$$

while $\varphi_2(x)$ has negative slope over $[x_2, x_3]$ and corresponds to setting $i = 2$ in (54),

$$\varphi_2(x) = 1 - (x - x_2)/h\,.$$

We can now easily integrate,

$$A_{2,3} = \int_\Omega \varphi_2\varphi_3\,\mathrm{d}x = \int_{x_2}^{x_3} \left(1 - \frac{x - x_2}{h}\right)\frac{x - x_2}{h}\,\mathrm{d}x = \frac{h}{6}\,.$$

The diagonal entry in the coefficient matrix becomes

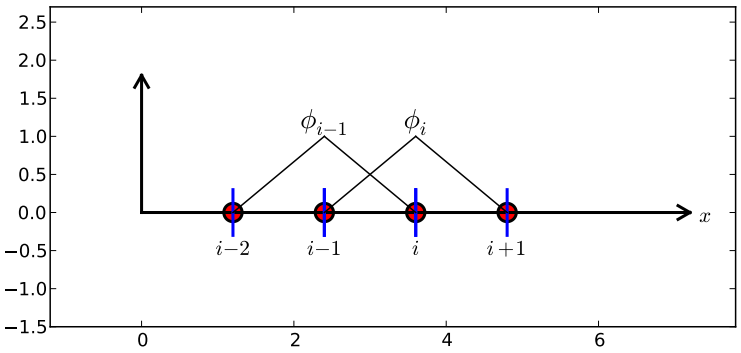

Figure 23: Illustration of two neighboring linear (hat) functions with general node numbers.

$$A_{2,2} = \int_{x_1}^{x_2} \left(\frac{x - x_1}{h}\right)^2 \mathrm{d}x + \int_{x_2}^{x_3} \left(1 - \frac{x - x_2}{h}\right)^2 \mathrm{d}x = \frac{2h}{3}\,.$$

The entry $A_{2,1}$ has an the integral that is geometrically similar to the situation in Figure 22, so we get $A_{2,1} = h/6$.

Calculating a general row in the matrix. We can now generalize the calculation of matrix entries to a general row number i. The entry $A_{i,i-1} = \int_\Omega \varphi_i \varphi_{i-1}\,\mathrm{d}x$ involves hat functions as depicted in Figure 23. Since the integral is geometrically identical to the situation with specific nodes 2 and 3, we realize that $A_{i,i-1} = A_{i,i+1} = h/6$ and $A_{i,i} = 2h/3$. However, we can compute the integral directly too:

$$\begin{aligned} A_{i,i-1} &= \int_\Omega \varphi_i \varphi_{i-1}\,\mathrm{d}x \\ &= \underbrace{\int_{x_{i-2}}^{x_{i-1}} \varphi_i \varphi_{i-1}\,\mathrm{d}x}_{\varphi_i = 0} + \int_{x_{i-1}}^{x_i} \varphi_i \varphi_{i-1}\,\mathrm{d}x + \underbrace{\int_{x_i}^{x_{i+1}} \varphi_i \varphi_{i-1}\,\mathrm{d}x}_{\varphi_{i-1} = 0} \\ &= \int_{x_{i-1}}^{x_i} \underbrace{\left(\frac{x - x_i}{h}\right)}_{\varphi_i(x)} \underbrace{\left(1 - \frac{x - x_{i-1}}{h}\right)}_{\varphi_{i-1}(x)} \mathrm{d}x = \frac{h}{6}\,. \end{aligned}$$

The particular formulas for $\varphi_{i-1}(x)$ and $\varphi_i(x)$ on $[x_{i-1}, x_i]$ are found from (54): φ_i is the linear function with positive slope, corresponding to the interval $[x_{i-1}, x_i]$ in (54), while ϕ_{i-1} has a negative slope so the definition in interval $[x_i, x_{i+1}]$ in (54) must be used. (The appearance of i in (54) and the integral might be confusing, as we speak about two different i indices.)

The first and last row of the coefficient matrix lead to slightly different integrals:

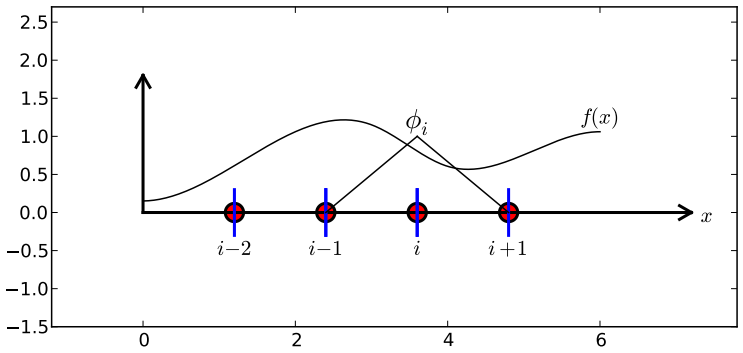

Figure 24: Right-hand side integral with the product of a basis function and the given function to approximate.

$$A_{0,0} = \int_\Omega \varphi_0^2 \,\mathrm{d}x = \int_{x_0}^{x_1} \left(1 - \frac{x - x_0}{h}\right)^2 \,\mathrm{d}x = \frac{h}{3}\,.$$

Similarly, $A_{N,N}$ involves an integral over only one element and equals hence $h/3$.

The general formula for b_i, see Figure 24, is now easy to set up

$$b_i = \int_\Omega \varphi_i(x) f(x)\,\mathrm{d}x = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} f(x)\,\mathrm{d}x + \int_{x_i}^{x_{i+1}} \left(1 - \frac{x - x_i}{h}\right) f(x)\,\mathrm{d}x\,. \tag{56}$$

We need a specific $f(x)$ function to compute these integrals. With $f(x) = x(1-x)$ and two equal-sized elements in $\Omega = [0, 1]$, one gets

$$A = \frac{h}{6}\begin{pmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 2 \end{pmatrix}, \quad b = \frac{h^2}{12}\begin{pmatrix} 2 - h \\ 12 - 14h \\ 10 - 17h \end{pmatrix}\,.$$

The solution becomes

$$c_0 = \frac{h^2}{6}, \quad c_1 = h - \frac{5}{6}h^2, \quad c_2 = 2h - \frac{23}{6}h^2\,.$$

The resulting function

$$u(x) = c_0\varphi_0(x) + c_1\varphi_1(x) + c_2\varphi_2(x)$$

is displayed in Figure 25 (left). Doubling the number of elements to four leads to the improved approximation in the right part of Figure 25.

3.7 Assembly of elementwise computations

The integrals above are naturally split into integrals over individual elements since the formulas change with the elements. This idea of splitting the integral is fundamental in all practical implementations of the finite element method.

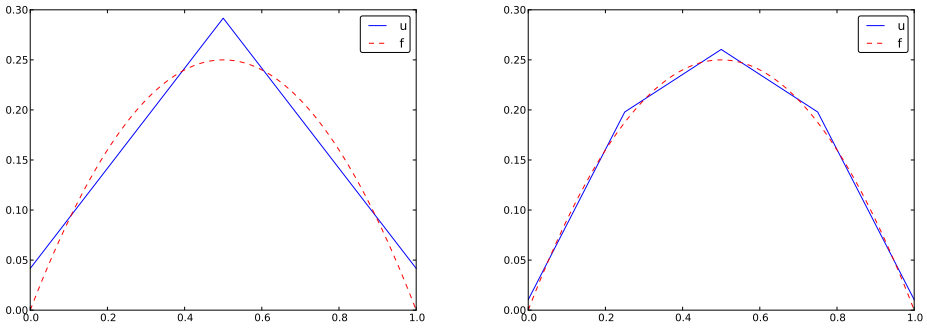

Figure 25: Least squares approximation of a parabola using 2 (left) and 4 (right) P1 elements.

Let us split the integral over Ω into a sum of contributions from each element:

$$A_{i,j} = \int_\Omega \varphi_i\varphi_j \,\mathrm{d}x = \sum_e A^{(e)}_{i,j}, \quad A^{(e)}_{i,j} = \int_{\Omega^{(e)}} \varphi_i\varphi_j \,\mathrm{d}x\,. \tag{57}$$

Now, $A^{(e)}_{i,j} \neq 0$ if and only if i and j are nodes in element e (look at Figure 23 to realize this property). Introduce $i = q(e,r)$ as the mapping of local node number r in element e to the global node number i. This is just a short mathematical notation for the expression `i=elements[e][r]` in a program. Let r and s be the local node numbers corresponding to the global node numbers $i = q(e,r)$ and $j = q(e,s)$. With d nodes per element, all the nonzero elements in $A^{(e)}_{i,j}$ arise from the integrals involving basis functions with indices corresponding to the global node numbers in element number e:

$$\int_{\Omega^{(e)}} \varphi_{q(e,r)}\varphi_{q(e,s)} \,\mathrm{d}x, \quad r,s = 0,\ldots,d\,.$$

These contributions can be collected in a $(d+1)\times(d+1)$ matrix known as the *element matrix*. Let $I_d = \{0,\ldots,d\}$ be the valid indices of r and s. We introduce the notation

$$\tilde{A}^{(e)} = \{\tilde{A}^{(e)}_{r,s}\}, \quad r,s \in I_d,$$

for the element matrix. For the case $d = 2$ we have

$$\tilde{A}^{(e)} = \begin{bmatrix} \tilde{A}^{(e)}_{0,0} & \tilde{A}^{(e)}_{0,1} & \tilde{A}^{(e)}_{0,2} \\ \tilde{A}^{(e)}_{1,0} & \tilde{A}^{(e)}_{1,1} & \tilde{A}^{(e)}_{1,2} \\ \tilde{A}^{(e)}_{2,0} & \tilde{A}^{(e)}_{2,1} & \tilde{A}^{(e)}_{2,2} \end{bmatrix}.$$

Given the numbers $\tilde{A}^{(e)}_{r,s}$, we should according to (57) add the contributions to the global coefficient matrix by

$$A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}^{(e)}_{r,s}, \quad r,s \in I_d\,. \tag{58}$$

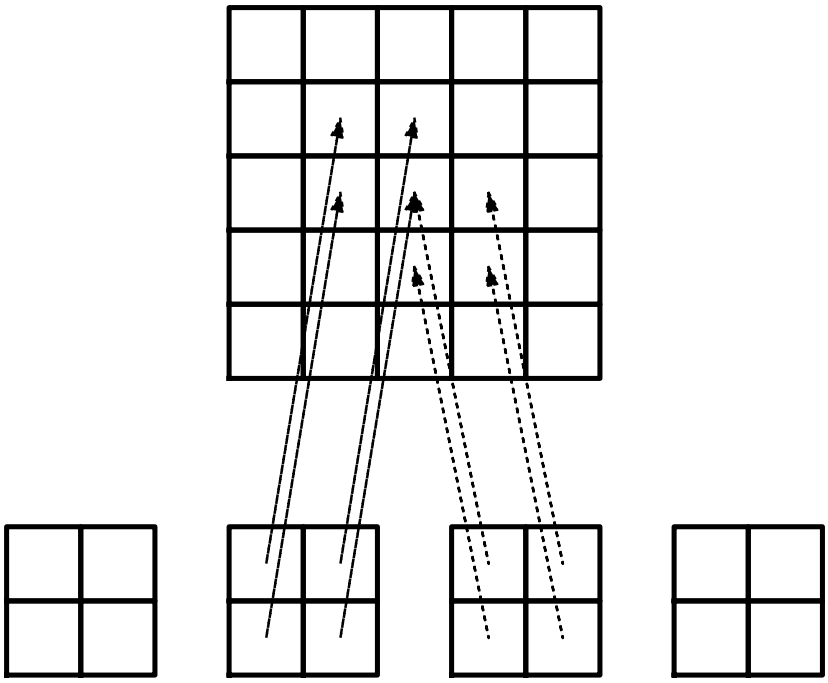

Figure 26: Illustration of matrix assembly: regularly numbered P1 elements.

This process of adding in elementwise contributions to the global matrix is called *finite element assembly* or simply *assembly*. Figure 26 illustrates how element matrices for elements with two nodes are added into the global matrix. More specifically, the figure shows how the element matrix associated with elements 1 and 2 assembled, assuming that global nodes are numbered from left to right in the domain. With regularly numbered P3 elements, where the element matrices have size 4×4, the assembly of elements 1 and 2 are sketched in Figure 27.

After assembly of element matrices corresponding to regularly numbered elements and nodes are understood, it is wise to study the assembly process for irregularly numbered elements and nodes. Figure 17 shows a mesh where the `elements` array, or $q(e, r)$ mapping in mathematical notation, is given as

```
elements = [[2, 1], [4, 5], [0, 4], [3, 0], [5, 2]]
```

The associated assembly of element matrices 1 and 2 is sketched in Figure 28.

These three assembly processes can also be animated[6].

The right-hand side of the linear system is also computed elementwise:

$$b_i = \int_\Omega f(x)\varphi_i(x)\,\mathrm{d}x = \sum_e b_i^{(e)}, \quad b_i^{(e)} = \int_{\Omega^{(e)}} f(x)\varphi_i(x)\,\mathrm{d}x\,. \tag{59}$$

[6] `http://tinyurl.com/opdfafk/pub/mov-approx/fe_assembly.html`

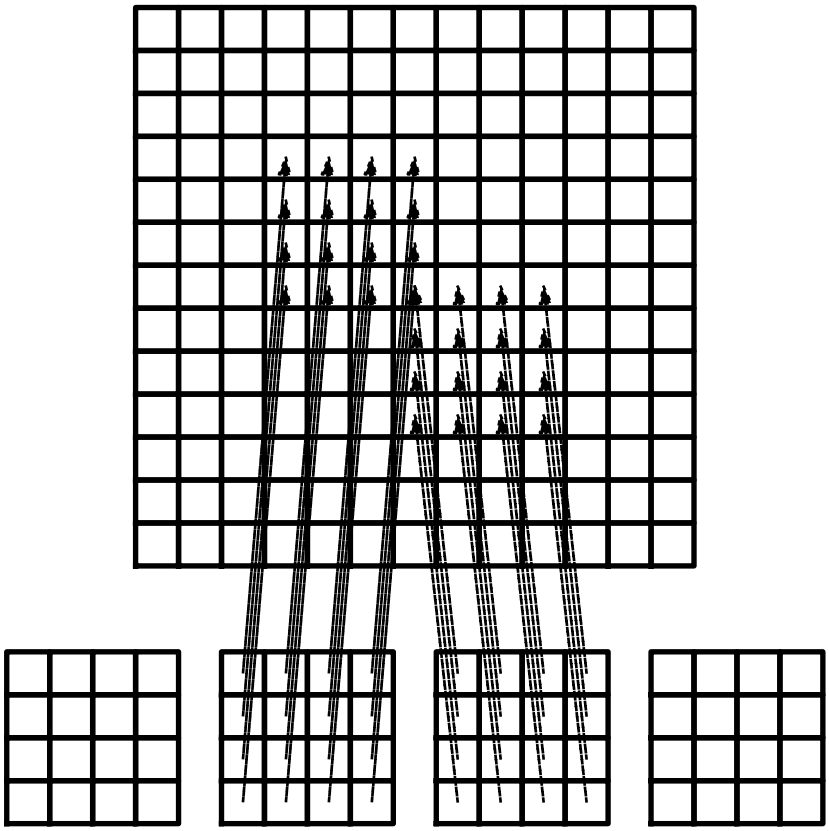

Figure 27: Illustration of matrix assembly: regularly numbered P3 elements.

We observe that $b_i^{(e)} \neq 0$ if and only if global node i is a node in element e (look at Figure 24 to realize this property). With d nodes per element we can collect the $d+1$ nonzero contributions $b_i^{(e)}$, for $i = q(e,r)$, $r \in I_d$, in an *element vector*

$$\tilde{b}_r^{(e)} = \{\tilde{b}_r^{(e)}\}, \quad r \in I_d\,.$$

These contributions are added to the global right-hand side by an assembly process similar to that for the element matrices:

$$b_{q(e,r)} := b_{q(e,r)} + \tilde{b}_r^{(e)}, \quad r \in I_d\,. \tag{60}$$

3.8 Mapping to a reference element

Instead of computing the integrals

$$\tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)\,\mathrm{d}x$$

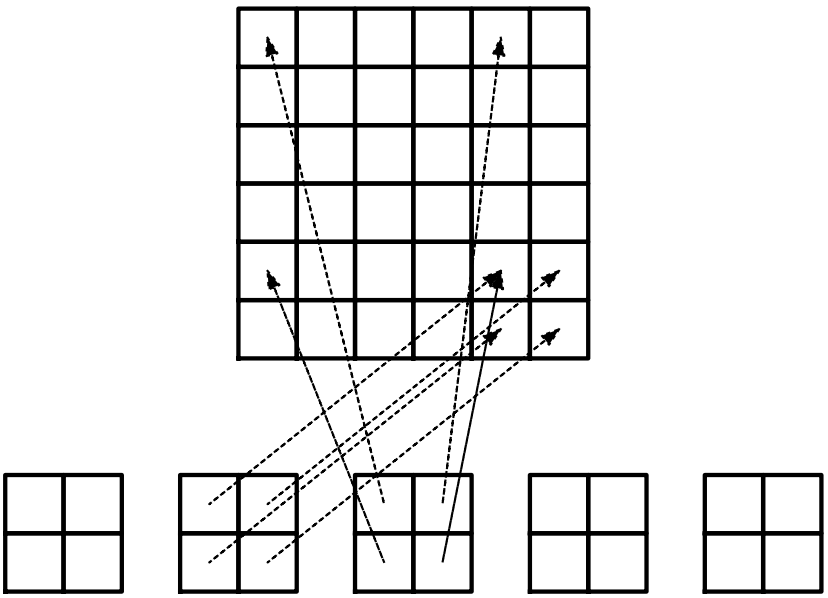

Figure 28: Illustration of matrix assembly: irregularly numbered P1 elements.

over some element $\Omega^{(e)} = [x_L, x_R]$, it is convenient to map the element domain $[x_L, x_R]$ to a standardized reference element domain $[-1, 1]$. (We have now introduced x_L and x_R as the left and right boundary points of an arbitrary element. With a natural, regular numbering of nodes and elements from left to right through the domain, we have $x_L = x_e$ and $x_R = x_{e+1}$ for P1 elements.)

Let $X \in [-1, 1]$ be the coordinate in the reference element. A linear or *affine mapping* from X to x reads

$$x = \frac{1}{2}(x_L + x_R) + \frac{1}{2}(x_R - x_L)X\,. \tag{61}$$

This relation can alternatively be expressed by

$$x = x_m + \frac{1}{2}hX, \tag{62}$$

where we have introduced the element midpoint $x_m = (x_L + x_R)/2$ and the element length $h = x_R - x_L$.

Integrating on the reference element is a matter of just changing the integration variable from x to X. Let

$$\tilde{\varphi}_r(X) = \varphi_{q(e,r)}(x(X)) \tag{63}$$

be the basis function associated with local node number r in the reference element. The integral transformation reads

$$\tilde{A}^{(e)}_{r,s} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)\,\mathrm{d}x = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\frac{dx}{dX}\,\mathrm{d}X\,. \tag{64}$$

The stretch factor dx/dX between the x and X coordinates becomes the determinant of the Jacobian matrix of the mapping between the coordinate systems in 2D and 3D. To obtain a uniform notation for 1D, 2D, and 3D problems we therefore replace dx/dX by $\det J$ already now. In 1D, $\det J = dx/dX = h/2$. The integration over the reference element is then written as

$$\tilde{A}^{(e)}_{r,s} = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X)\det J\,dX\,. \tag{65}$$

The corresponding formula for the element vector entries becomes

$$\tilde{b}^{(e)}_{r} = \int_{\Omega^{(e)}} f(x)\varphi_{q(e,r)}(x)dx = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\det J\,dX\,. \tag{66}$$

Since we from now on will work in the reference element, we need explicit mathematical formulas for the basis functions $\varphi_i(x)$ in the reference element only, i.e., we only need to specify formulas for $\tilde{\varphi}_r(X)$. This is a very convenient simplification compared to specifying piecewise polynomials in the physical domain.

The $\tilde{\varphi}_r(x)$ functions are simply the Lagrange polynomials defined through the local nodes in the reference element. For $d = 1$ and two nodes per element, we have the linear Lagrange polynomials

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X) \tag{67}$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X) \tag{68}$$

Quadratic polynomials, $d = 2$, have the formulas

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X - 1)X \tag{69}$$

$$\tilde{\varphi}_1(X) = 1 - X^2 \tag{70}$$

$$\tilde{\varphi}_2(X) = \frac{1}{2}(X + 1)X \tag{71}$$

In general,

$$\tilde{\varphi}_r(X) = \prod_{s=0,s\neq r}^{d} \frac{X - X_{(s)}}{X_{(r)} - X_{(s)}}, \tag{72}$$

where $X_{(0)}, \ldots, X_{(d)}$ are the coordinates of the local nodes in the reference element. These are normally uniformly spaced: $X_{(r)} = -1 + 2r/d$, $r \in I_d$.

Why reference elements?

The great advantage of using reference elements is that the formulas for the basis functions, $\tilde{\varphi}_r(X)$, are the same for all elements and independent of the element geometry (length and location in the mesh). The geometric information is "factored out" in the simple mapping formula and the associated $\det J$ quantity, but this information is (here taken as) the same for element types. Also, the integration domain is the same for all elements.

3.9 Example: Integration over a reference element

To illustrate the concepts from the previous section in a specific example, we now consider calculation of the element matrix and vector for a specific choice of d and $f(x)$. A simple choice is $d = 1$ (P1 elements) and $f(x) = x(1-x)$ on $\Omega = [0,1]$. We have the general expressions (65) and (66) for $\tilde{A}^{(e)}_{r,s}$ and $\tilde{b}^{(e)}_r$. Writing these out for the choices (67) and (68), and using that $\det J = h/2$, we can do the following calculations of the element matrix entries:

$$\tilde{A}^{(e)}_{0,0} = \int_{-1}^{1} \tilde{\varphi}_0(X)\tilde{\varphi}_0(X)\frac{h}{2}dX$$
$$= \int_{-1}^{1} \frac{1}{2}(1-X)\frac{1}{2}(1-X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1-X)^2dX = \frac{h}{3}, \tag{73}$$
$$\tilde{A}^{(e)}_{1,0} = \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_0(X)\frac{h}{2}dX$$
$$= \int_{-1}^{1} \frac{1}{2}(1+X)\frac{1}{2}(1-X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1-X^2)dX = \frac{h}{6}, \tag{74}$$
$$\tilde{A}^{(e)}_{0,1} = \tilde{A}^{(e)}_{1,0}, \tag{75}$$
$$\tilde{A}^{(e)}_{1,1} = \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_1(X)\frac{h}{2}dX$$
$$= \int_{-1}^{1} \frac{1}{2}(1+X)\frac{1}{2}(1+X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1+X)^2dX = \frac{h}{3}\,. \tag{76}$$

The corresponding entries in the element vector becomes

$$
\begin{aligned}
\tilde{b}_0^{(e)} &= \int_{-1}^{1} f(x(X))\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX))\frac{1}{2}(1-X)\frac{h}{2}dX \\
&= -\frac{1}{24}h^3 + \frac{1}{6}h^2 x_m - \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m \qquad (77) \\
\tilde{b}_1^{(e)} &= \int_{-1}^{1} f(x(X))\tilde{\varphi}_1(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX))\frac{1}{2}(1+X)\frac{h}{2}dX \\
&= -\frac{1}{24}h^3 - \frac{1}{6}h^2 x_m + \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m\,. \qquad (78)
\end{aligned}
$$

In the last two expressions we have used the element midpoint x_m.

Integration of lower-degree polynomials above is tedious, and higher-degree polynomials involve very much more algebra, but `sympy` may help. For example, we can easily calculate (73), (73), and (77) by

```
>>> import sympy as sp
>>> x, x_m, h, X = sp.symbols('x x_m h X')
>>> sp.integrate(h/8*(1-X)**2, (X, -1, 1))
h/3
>>> sp.integrate(h/8*(1+X)*(1-X), (X, -1, 1))
h/6
>>> x = x_m + h/2*X
>>> b_0 = sp.integrate(h/4*x*(1-x)*(1-X), (X, -1, 1))
>>> print b_0
-h**3/24 + h**2*x_m/6 - h**2/12 - h*x_m**2/2 + h*x_m/2
```

For inclusion of formulas in documents (like the present one), `sympy` can print expressions in LaTeX format:

```
>>> print sp.latex(b_0, mode='plain')
- \frac{1}{24} h^{3} + \frac{1}{6} h^{2} x_{m}
- \frac{1}{12} h^{2} - \half h x_{m}^{2}
+ \half h x_{m}
```

4 Implementation

Based on the experience from the previous example, it makes sense to write some code to automate the analytical integration process for any choice of finite element basis functions. In addition, we can automate the assembly process and linear system solution. Appropriate functions for this purpose document all details of all steps in the finite element computations and can be found in the

module file `fe_approx1D.py`[7]. The key steps in the computational machinery are now explained in detail in terms of code and text.

4.1 Integration

First we need a Python function for defining $\tilde{\varphi}_r(X)$ in terms of a Lagrange polynomial of degree `d`:

```
import sympy as sp
import numpy as np

def phi_r(r, X, d):
    if isinstance(X, sp.Symbol):
        h = sp.Rational(1, d)  # node spacing
        nodes = [2*i*h - 1 for i in range(d+1)]
    else:
        # assume X is numeric: use floats for nodes
        nodes = np.linspace(-1, 1, d+1)
    return Lagrange_polynomial(X, r, nodes)

def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p
```

Observe how we construct the `phi_r` function to be a symbolic expression for $\tilde{\varphi}_r(X)$ if `X` is a `Symbol` object from `sympy`. Otherwise, we assume that `X` is a `float` object and compute the corresponding floating-point value of $\tilde{\varphi}_r(X)$. Recall that the `Lagrange_polynomial` function, here simply copied from Section 2.7, works with both symbolic and numeric variables.

The complete basis $\tilde{\varphi}_0(X), \ldots, \tilde{\varphi}_d(X)$ on the reference element, represented as a list of symbolic expressions, is constructed by

```
def basis(d=1):
    X = sp.Symbol('X')
    phi = [phi_r(r, X, d) for r in range(d+1)]
    return phi
```

Now we are in a position to write the function for computing the element matrix:

```
def element_matrix(phi, Omega_e, symbolic=True):
    n = len(phi)
    A_e = sp.zeros((n, n))
    X = sp.Symbol('X')
    if symbolic:
        h = sp.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    detJ = h/2  # dx/dX
    for r in range(n):
```

[7] `http://tinyurl.com/nm5587k/fem/fe_approx1D.py`

```
        for s in range(r, n):
            A_e[r,s] = sp.integrate(phi[r]*phi[s]*detJ, (X, -1, 1))
            A_e[s,r] = A_e[r,s]
    return A_e
```

In the symbolic case (`symbolic` is `True`), we introduce the element length as a symbol `h` in the computations. Otherwise, the real numerical value of the element interval `Omega_e` is used and the final matrix elements are numbers, not symbols. This functionality can be demonstrated:

```
>>> from fe_approx1D import *
>>> phi = basis(d=1)
>>> phi
[1/2 - X/2, 1/2 + X/2]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=True)
[h/3, h/6]
[h/6, h/3]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=False)
[0.0333333333333333, 0.0166666666666667]
[0.0166666666666667, 0.0333333333333333]
```

The computation of the element vector is done by a similar procedure:

```
def element_vector(f, phi, Omega_e, symbolic=True):
    n = len(phi)
    b_e = sp.zeros((n, 1))
    # Make f a function of X
    X = sp.Symbol('X')
    if symbolic:
        h = sp.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    x = (Omega_e[0] + Omega_e[1])/2 + h/2*X  # mapping
    f = f.subs('x', x)  # substitute mapping formula for x
    detJ = h/2  # dx/dX
    for r in range(n):
        b_e[r] = sp.integrate(f*phi[r]*detJ, (X, -1, 1))
    return b_e
```

Here we need to replace the symbol `x` in the expression for `f` by the mapping formula such that `f` can be integrated in terms of X, cf. the formula $\tilde{b}_r^{(e)} = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\frac{h}{2}dX$.

The integration in the element matrix function involves only products of polynomials, which `sympy` can easily deal with, but for the right-hand side `sympy` may face difficulties with certain types of expressions `f`. The result of the integral is then an `Integral` object and not a number or expression as when symbolic integration is successful. It may therefore be wise to introduce a fallback on numerical integration. The symbolic integration can also take much time before an unsuccessful conclusion so we may also introduce a parameter `symbolic` and set it to `False` in order to avoid symbolic integration:

```
def element_vector(f, phi, Omega_e, symbolic=True):
        ...
        if symbolic:
            I = sp.integrate(f*phi[r]*detJ, (X, -1, 1))
        if not symbolic or isinstance(I, sp.Integral):
            h = Omega_e[1] - Omega_e[0]  # Ensure h is numerical
            detJ = h/2
            integrand = sp.lambdify([X], f*phi[r]*detJ)
            I = sp.mpmath.quad(integrand, [-1, 1])
        b_e[r] = I
        ...
```

Numerical integration requires that the symbolic integrand is converted to a plain Python function (`integrand`) and that the element length `h` is a real number.

4.2 Linear system assembly and solution

The complete algorithm for computing and assembling the elementwise contributions takes the following form

```
def assemble(nodes, elements, phi, f, symbolic=True):
    N_n, N_e = len(nodes), len(elements)
    if symbolic:
        A = sp.zeros((N_n, N_n))
        b = sp.zeros((N_n, 1))    # note: (N_n, 1) matrix
    else:
        A = np.zeros((N_n, N_n))
        b = np.zeros(N_n)
    for e in range(N_e):
        Omega_e = [nodes[elements[e][0]], nodes[elements[e][-1]]]

        A_e = element_matrix(phi, Omega_e, symbolic)
        b_e = element_vector(f, phi, Omega_e, symbolic)

        for r in range(len(elements[e])):
            for s in range(len(elements[e])):
                A[elements[e][r],elements[e][s]] += A_e[r,s]
            b[elements[e][r]] += b_e[r]
    return A, b
```

The `nodes` and `elements` variables represent the finite element mesh as explained earlier.

Given the coefficient matrix `A` and the right-hand side `b`, we can compute the coefficients $\{c_i\}_{i\in\mathcal{I}_s}$ in the expansion $u(x) = \sum_j c_j\varphi_j$ as the solution vector `c` of the linear system:

```
if symbolic:
    c = A.LUsolve(b)
else:
    c = np.linalg.solve(A, b)
```

When `A` and `b` are `sympy` arrays, the solution procedure implied by `A.LUsolve` is symbolic. Otherwise, `A` and `b` are `numpy` arrays and a standard numerical solver is called. The symbolic version is suited for small problems only (small N values)

since the calculation time becomes prohibitively large otherwise. Normally, the symbolic *integration* will be more time consuming in small problems than the symbolic *solution* of the linear system.

4.3 Example on computing symbolic approximations

We can exemplify the use of `assemble` on the computational case from Section 3.6 with two P1 elements (linear basis functions) on the domain $\Omega = [0, 1]$. Let us first work with a symbolic element length:

```
>>> h, x = sp.symbols('h x')
>>> nodes = [0, h, 2*h]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,   0]
[h/6, 2*h/3, h/6]
[  0,   h/6, h/3]
>>> b
[     h**2/6 - h**3/12]
[      h**2 - 7*h**3/6]
[5*h**2/6 - 17*h**3/12]
>>> c = A.LUsolve(b)
>>> c
[                            h**2/6]
[12*(7*h**2/12 - 35*h**3/72)/(7*h)]
[  7*(4*h**2/7 - 23*h**3/21)/(2*h)]
```

4.4 Comparison with finite elements and interpolation/-collocation

We may, for comparison, compute the `c` vector corresponding to an interpolation/collocation method with finite element basis functions. Choosing the nodes as points, the principle is

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x_i) = f(x_i), \quad i \in \mathcal{I}_s \,.$$

The coefficient matrix $A_{i,j} = \varphi_j(x_i)$ becomes the identity matrix because basis function number j vanishes at all nodes, except node j: $\varphi_j(x_i) = \delta_{ij}$. Therefore, $c_i = f(x_i)$.

The associated `sympy` calculations are

```
>>> fn = sp.lambdify([x], f)
>>> c = [fn(xc) for xc in nodes]
>>> c
[0, h*(1 - h), 2*h*(1 - 2*h)]
```

These expressions are much simpler than those based on least squares or projection in combination with finite element basis functions.

4.5 Example on computing numerical approximations

The numerical computations corresponding to the symbolic ones in Section 4.3, and still done by `sympy` and the `assemble` function, go as follows:

```
>>> nodes = [0, 0.5, 1]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> x = sp.Symbol('x')
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=False)
>>> A
[ 0.166666666666667, 0.0833333333333333,                  0]
[0.0833333333333333,  0.333333333333333, 0.0833333333333333]
[                 0, 0.0833333333333333,  0.166666666666667]
>>> b
[          0.03125]
[0.104166666666667]
[          0.03125]
>>> c = A.LUsolve(b)
>>> c
[0.0416666666666666]
[ 0.291666666666667]
[0.0416666666666666]
```

The `fe_approx1D` module contains functions for generating the `nodes` and `elements` lists for equal-sized elements with any number of nodes per element. The coordinates in `nodes` can be expressed either through the element length symbol `h` (`symbolic=True`) or by real numbers (`symbolic=False`):

```
nodes, elements = mesh_uniform(N_e=10, d=3, Omega=[0,1],
                               symbolic=True)
```

There is also a function

```
def approximate(f, symbolic=False, d=1, N_e=4, filename='tmp.pdf'):
```

which computes a mesh with `N_e` elements, basis functions of degree `d`, and approximates a given symbolic expression `f` by a finite element expansion $u(x) = \sum_j c_j\varphi_j(x)$. When `symbolic` is `False`, $u(x) = \sum_j c_j\varphi_j(x)$ can be computed at a (large) number of points and plotted together with $f(x)$. The construction of u points from the solution vector `c` is done elementwise by evaluating $\sum_r c_r\tilde{\varphi}_r(X)$ at a (large) number of points in each element in the local coordinate system, and the discrete (x, u) values on each element are stored in separate arrays that are finally concatenated to form a global array for x and for u. The details are found in the `u_glob` function in `fe_approx1D.py`.

4.6 The structure of the coefficient matrix

Let us first see how the global matrix looks like if we assemble symbolic element matrices, expressed in terms of `h`, from several elements:

```
>>> d=1; N_e=8; Omega=[0,1]  # 8 linear elements on [0,1]
>>> phi = basis(d)
>>> f = x*(1-x)
>>> nodes, elements = mesh_symbolic(N_e, d, Omega)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,     0,     0,     0,     0,     0,     0,   0]
[h/6, 2*h/3,   h/6,     0,     0,     0,     0,     0,   0]
[  0,   h/6, 2*h/3,   h/6,     0,     0,     0,     0,   0]
[  0,     0,   h/6, 2*h/3,   h/6,     0,     0,     0,   0]
[  0,     0,     0,   h/6, 2*h/3,   h/6,     0,     0,   0]
[  0,     0,     0,     0,   h/6, 2*h/3,   h/6,     0,   0]
[  0,     0,     0,     0,     0,   h/6, 2*h/3,   h/6,   0]
[  0,     0,     0,     0,     0,     0,   h/6, 2*h/3, h/6]
[  0,     0,     0,     0,     0,     0,     0,   h/6, h/3]
```

The reader is encouraged to assemble the element matrices by hand and verify this result, as this exercise will give a hands-on understanding of what the assembly is about. In general we have a coefficient matrix that is tridiagonal:

$$A = \frac{h}{6}\begin{pmatrix} 2 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 1 & 4 & 1 & \ddots & & & & & \vdots \\ 0 & 1 & 4 & 1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & 1 & 4 & 1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & 1 & 4 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 & 2 \end{pmatrix} \tag{79}$$

The structure of the right-hand side is more difficult to reveal since it involves an assembly of elementwise integrals of $f(x(X))\tilde{\varphi}_r(X)h/2$, which obviously depend on the particular choice of $f(x)$. Numerical integration can give some insight into the nature of the right-hand side. For this purpose it is easier to look at the integration in x coordinates, which gives the general formula (56). For equal-sized elements of length h, we can apply the Trapezoidal rule at the global node points to arrive at

$$b_i = h\left(\frac{1}{2}\varphi_i(x_0)f(x_0) + \frac{1}{2}\varphi_i(x_N)f(x_N) + \sum_{j=1}^{N-1}\varphi_i(x_j)f(x_j)\right) \tag{80}$$

$$= \begin{cases} \frac{1}{2}hf(x_i), & i = 0 \text{ or } i = N, \\ hf(x_i), & 1 \leq i \leq N-1 \end{cases} \tag{81}$$

The reason for this simple formula is simply that φ_i is either 0 or 1 at the nodes and 0 at all but one of them.

Going to P2 elements (`d=2`) leads to the element matrix

$$A^{(e)} = \frac{h}{30}\begin{pmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{pmatrix} \tag{82}$$

and the following global assembled matrix from four elements:

$$A = \frac{h}{30}\begin{pmatrix} 4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 16 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & 8 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 16 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 8 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 16 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & 8 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 16 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 4 \end{pmatrix} \tag{83}$$

In general, for i odd we have the nonzeroes

$$A_{i,i-2} = -1, \quad A_{i-1,i} = 2, \quad A_{i,i} = 8, \quad A_{i+1,i} = 2, \quad A_{i+2,i} = -1,$$

multiplied by $h/30$, and for i even we have the nonzeros

$$A_{i-1,i} = 2, \quad A_{i,i} = 16, \quad A_{i+1,i} = 2,$$

multiplied by $h/30$. The rows with odd numbers correspond to nodes at the element boundaries and get contributions from two neighboring elements in the assembly process, while the even numbered rows correspond to internal nodes in the elements where the only one element contributes to the values in the global matrix.

4.7 Applications

With the aid of the `approximate` function in the `fe_approx1D` module we can easily investigate the quality of various finite element approximations to some given functions. Figure 29 shows how linear and quadratic elements approximates the polynomial $f(x) = x(1-x)^8$ on $\Omega = [0, 1]$, using equal-sized elements. The results arise from the program

```
import sympy as sp
from fe_approx1D import approximate
x = sp.Symbol('x')

approximate(f=x*(1-x)**8, symbolic=False, d=1, N_e=4)
approximate(f=x*(1-x)**8, symbolic=False, d=2, N_e=2)
approximate(f=x*(1-x)**8, symbolic=False, d=1, N_e=8)
approximate(f=x*(1-x)**8, symbolic=False, d=2, N_e=4)
```

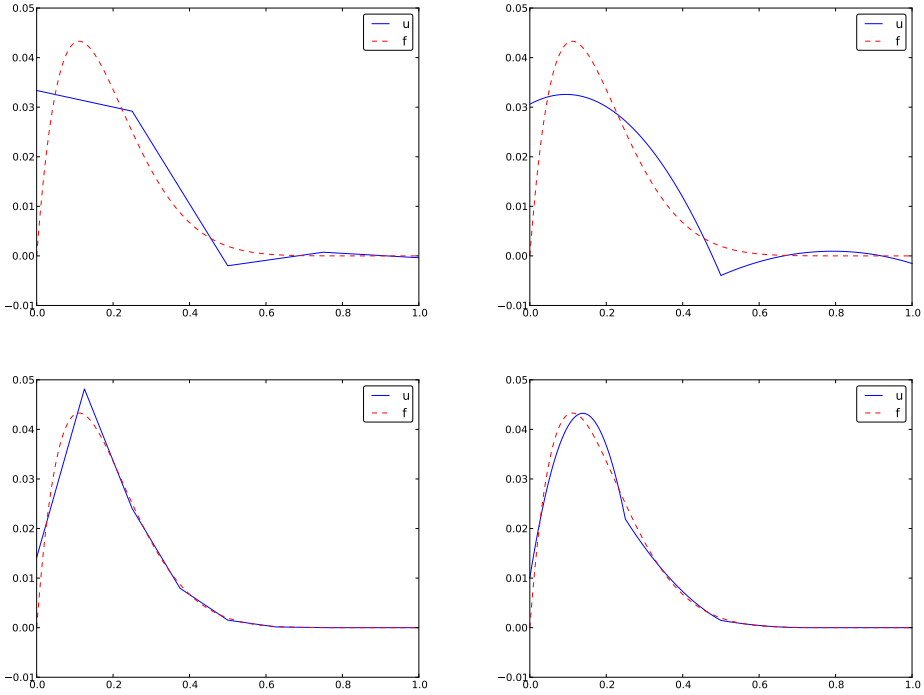

Figure 29: Comparison of the finite element approximations: 4 P1 elements with 5 nodes (upper left), 2 P2 elements with 5 nodes (upper right), 8 P1 elements with 9 nodes (lower left), and 4 P2 elements with 9 nodes (lower right).

The quadratic functions are seen to be better than the linear ones for the same value of N, as we increase N. This observation has some generality: higher degree is not necessarily better on a coarse mesh, but it is as we refined the mesh.

4.8 Sparse matrix storage and solution

Some of the examples in the preceding section took several minutes to compute, even on small meshes consisting of up to eight elements. The main explanation for slow computations is unsuccessful symbolic integration: `sympy` may use a lot of energy on integrals like $\int f(x(X))\tilde{\varphi}_r(X)h/2dx$ before giving up, and the program then resorts to numerical integration. Codes that can deal with a large number of basis functions and accept flexible choices of $f(x)$ should compute all integrals numerically and replace the matrix objects from `sympy` by the far more efficient array objects from `numpy`.

Another reason for slow code is related to the fact that most of the matrix entries $A_{i,j}$ are zero, because $(\varphi_i, \varphi_j) = 0$ unless i and j are nodes in the same element. A matrix whose majority of entries are zeros, is known as a *sparse*

Figure 30: Matrix sparsity pattern for left-to-right numbering (left) and random numbering (right) of nodes in P1 elements.

Figure 31: Matrix sparsity pattern for left-to-right numbering (left) and random numbering (right) of nodes in P3 elements.

matrix. The sparsity should be utilized in software as it dramatically decreases the storage demands and the CPU-time needed to compute the solution of the linear system. This optimization is not critical in 1D problems where modern computers can afford computing with all the zeros in the complete square matrix, but in 2D and especially in 3D, sparse matrices are fundamental for feasible finite element computations.

In 1D problems, using a numbering of nodes and elements from left to right over the domain, the assembled coefficient matrix has only a few diagonals different from zero. More precisely, $2d + 1$ diagonals are different from zero. With a different numbering of global nodes, say a random ordering, the diagonal structure is lost, but the number of nonzero elements is unaltered. Figures 30 and 31 exemplify sparsity patterns.

The `scipy.sparse` library supports creation of sparse matrices and linear system solution.

- `scipy.sparse.diags` for matrix defined via diagonals
- `scipy.sparse.lil_matrix` for creation via setting matrix entries
- `scipy.sparse.dok_matrix` for creation via setting matrix entries

5 Comparison of finite element and finite difference approximation

The previous sections on approximating f by a finite element function u utilize the projection/Galerkin or least squares approaches to minimize the approximation error. We may, alternatively, use the collocation/interpolation method as described in Section 4.4. Here we shall compare these three approaches with what one does in the finite difference method when representing a given function on a mesh.

5.1 Finite difference approximation of given functions

Approximating a given function $f(x)$ on a mesh in a finite difference context will typically just sample f at the mesh points. If u_i is the value of the approximate u at the mesh point x_i, we have $u_i = f(x_i)$. The collocation/interpolation method using finite element basis functions gives exactly the same representation, as shown Section 4.4,

$$u(x_i) = c_i = f(x_i)\,.$$

How does a finite element Galerkin or least squares approximation differ from this straightforward interpolation of f? This is the question to be addressed next. We now limit the scope to P1 elements since this is the element type that gives formulas closest to those arising in the finite difference method.

5.2 Finite difference interpretation of a finite element approximation

The linear system arising from a Galerkin or least squares approximation reads in general

$$\sum_{j\in\mathcal{I}_s} c_j(\psi_i,\psi_j) = (f,\psi_i), \quad i\in\mathcal{I}_s\,.$$

In the finite element approximation we choose $\psi_i = \varphi_i$. With φ_i corresponding to P1 elements and a uniform mesh of element length h we have in Section 3.6 calculated the matrix with entries (φ_i,φ_j). Equation number i reads

$$\frac{h}{6}(u_{i-1} + 4u_i + u_{i+1}) = (f,\varphi_i)\,. \tag{84}$$

The first and last equation, corresponding to $i = 0$ and $i = N$ are slightly different, see Section 4.6.

The finite difference counterpart to (84) is just $u_i = f_i$ as explained in Section 5.1. To easier compare this result to the finite element approach to approximating functions, we can rewrite the left-hand side of (84) as

$$h(u_i + \frac{1}{6}(u_{i-1} - 2u_i + u_{i+1}))\,. \tag{85}$$

Thinking in terms of finite differences, we can write this expression using finite difference operator notation:

$$[h(u + \frac{h^2}{6} D_x D_x u)]_i,$$

which is nothing but the standard discretization of

$$h(u + \frac{h^2}{6} u'') \,.$$

Before interpreting the approximation procedure as solving a differential equation, we need to work out what the right-hand side is in the context of P1 elements. Since φ_i is the linear function that is 1 at x_i and zero at all other nodes, only the interval $[x_{i-1}, x_{i+1}]$ contribute to the integral on the right-hand side. This integral is naturally split into two parts according to (54):

$$(f, \varphi_i) = \int_{x_{i-1}}^{x_i} f(x) \frac{1}{h}(x - x_{i-1}) dx + \int_{x_i}^{x_{i+1}} f(x)(1 - \frac{1}{h}(x - x_i)) dx \,.$$

However, if f is not known we cannot do much else with this expression. It is clear that many values of f around x_i contributes to the right-hand side, not just the single point value $f(x_i)$ as in the finite difference method.

To proceed with the right-hand side, we can turn to numerical integration schemes. The Trapezoidal method for (f, φ_i), based on sampling the integrand $f\varphi_i$ at the node points $x_i = ih$ gives

$$(f, \varphi_i) = \int_\Omega f\varphi_i dx \approx h\frac{1}{2}(f(x_0)\varphi_i(x_0) + f(x_N)\varphi_i(x_N)) + h\sum_{j=1}^{N-1} f(x_j)\varphi_i(x_j) \,.$$

Since φ_i is zero at all these points, except at x_i, the Trapezoidal rule collapses to one term:

$$(f, \varphi_i) \approx hf(x_i), \tag{86}$$

for $i = 1, \ldots, N-1$, which is the same result as with collocation/interpolation, and of course the same result as in the finite difference method. For the end points $i = 0$ and $i = N$ we get contribution from only one element so

$$(f, \varphi_i) \approx \frac{1}{2} hf(x_i), \quad i = 0, \ i = N \,. \tag{87}$$

Simpson's rule with sample points also in the middle of the elements, at $x_{i+\frac{1}{2}} = (x_i + x_{i+1})/2$, can be written as

$$\int_\Omega g(x)dx \approx \frac{\tilde{h}}{3}\left(g(x_0) + 2\sum_{j=1}^{N-1} g(x_j) + 4\sum_{j=0}^{N-1} g(x_{j+\frac{1}{2}}) + f(x_{2N}) \right),$$

where $\tilde{h} = h/2$ is the spacing between the sample points. Our integrand is $g = f\varphi_i$. For all the node points, $\varphi_i(x_j) = \delta_{ij}$, and therefore $\sum_{j=1}^{N-1} f(x_j)\varphi_i(x_j) = f(x_i)$. At the midpoints, $\varphi_i(x_{i\pm\frac{1}{2}}) = 1/2$ and $\varphi_i(x_{j+\frac{1}{2}}) = 0$ for $j > 1$ and $j < i - 1$. Consequently,

$$\sum_{j=0}^{N-1} f(x_{j+\frac{1}{2}})\varphi_i(x_{j+\frac{1}{2}}) = \frac{1}{2}(fx_{j-\frac{1}{2}} + x_{j+\frac{1}{2}})\,.$$

When $1 \leq i \leq N-1$ we then get

$$(f, \varphi_i) \approx \frac{h}{3}(f_{i-\frac{1}{2}} + f_i + f_{i+\frac{1}{2}})\,. \tag{88}$$

This result shows that, with Simpson's rule, the finite element method operates with the average of f over three points, while the finite difference method just applies f at one point. We may interpret this as a "smearing" or smoothing of f by the finite element method.

We can now summarize our findings. With the approximation of (f, φ_i) by the Trapezoidal rule, P1 elements give rise to equations that can be expressed as a finite difference discretization of

$$u + \frac{h^2}{6}u'' = f, \quad u'(0) = u'(L) = 0, \tag{89}$$

expressed with operator notation as

$$[u + \frac{h^2}{6}D_xD_xu = f]_i\,. \tag{90}$$

As $h \to 0$, the extra term proportional to u'' goes to zero, and the two methods are then equal.

With the Simpson's rule, we may say that we solve

$$[u + \frac{h^2}{6}D_xD_xu = \bar{f}]_i, \tag{91}$$

where $\bar{f}_i$ means the average $\frac{1}{3}(f_{i-1/2} + f_i + f_{i+1/2})$.

The extra term $\frac{h^2}{6}u''$ represents a smoothing effect: with just this term, we would find u by integrating f twice and thereby smooth f considerably. In addition, the finite element representation of f involves an average, or a smoothing, of f on the right-hand side of the equation system. If f is a noisy function, direct interpolation $u_i = f_i$ may result in a noisy u too, but with a Galerkin or least squares formulation and P1 elements, we should expect that u is smoother than f unless h is very small.

The interpretation that finite elements tend to smooth the solution is valid in applications far beyond approximation of 1D functions.

5.3 Making finite elements behave as finite differences

With a simple trick, using numerical integration, we can easily produce the result $u_i = f_i$ with the Galerkin or least square formulation with P1 elements. This is useful in many occasions when we deal with more difficult differential equations and want the finite element method to have properties like the finite difference method (solving standard linear wave equations is one primary example).

Computations in physical space. We have already seen that applying the Trapezoidal rule to the right-hand side (f, φ_i) simply gives f sampled at x_i. Using the Trapezoidal rule on the matrix entries $A_{i,j} = (\varphi_i, \varphi_j)$ involves a sum

$$\sum_k \varphi_i(x_k)\varphi_j(x_k),$$

but $\varphi_i(x_k) = \delta_{ik}$ and $\varphi_j(x_k) = \delta_{jk}$. The product $\varphi_i\varphi_j$ is then different from zero only when sampled at x_i and $i = j$. The Trapezoidal approximation to the integral is then

$$(\varphi_i, \varphi_j) \approx h, \quad i = j,$$

and zero if $i \neq j$. This means that we have obtained a diagonal matrix! The first and last diagonal elements, (φ_0, φ_0) and (φ_N, φ_N) get contribution only from the first and last element, respectively, resulting in the approximate integral value $h/2$. The corresponding right-hand side also has a factor $1/2$ for $i = 0$ and $i = N$. Therefore, the least squares or Galerkin approach with P1 elements and Trapezoidal integration results in

$$c_i = f_i, \quad i \in \mathcal{I}_s\,.$$

Simpsons's rule can be used to achieve a similar result for P2 elements, i.e, a diagonal coefficient matrix, but with the previously derived average of f on the right-hand side.

Elementwise computations. Identical results to those above will arise if we perform elementwise computations. The idea is to use the Trapezoidal rule on the reference element for computing the element matrix and vector. When assembled, the same equations $c_i = f(x_i)$ arise. Exercise 19 encourages you to carry out the details.

Terminology. The matrix with entries (φ_i, φ_j) typically arises from terms proportional to u in a differential equation where u is the unknown function. This matrix is often called the *mass matrix*, because in the early days of the finite element method, the matrix arose from the mass times acceleration term in Newton's second law of motion. Making the mass matrix diagonal by, e.g., numerical integration, as demonstrated above, is a widely used technique and is called *mass lumping*. In time-dependent problems it can sometimes enhance the

numerical accuracy and computational efficiency of the finite element method. However, there are also examples where mass lumping destroys accuracy.

6 A generalized element concept

So far, finite element computing has employed the `nodes` and `element` lists together with the definition of the basis functions in the reference element. Suppose we want to introduce a piecewise constant approximation with one basis function $\tilde{\varphi}_0(x) = 1$ in the reference element, corresponding to a $\varphi_i(x)$ function that is 1 on element number i and zero on all other elements. Although we could associate the function value with a node in the middle of the elements, there are no nodes at the ends, and the previous code snippets will not work because we cannot find the element boundaries from the `nodes` list.

6.1 Cells, vertices, and degrees of freedom

We now introduce *cells* as the subdomains $\Omega^{(e)}$ previously referred as elements. The cell boundaries are denoted as *vertices*. The reason for this name is that cells are recognized by their vertices in 2D and 3D. We also define a set of *degrees of freedom*, which are the quantities we aim to compute. The most common type of degree of freedom is the value of the unknown function u at some point. (For example, we can introduce nodes as before and say the degrees of freedom are the values of u at the nodes.) The basis functions are constructed so that they equal unity for one particular degree of freedom and zero for the rest. This property ensures that when we evaluate $u = \sum_j c_j\varphi_j$ for degree of freedom number i, we get $u = c_i$. Integrals are performed over cells, usually by mapping the cell of interest to a *reference cell*.

With the concepts of cells, vertices, and degrees of freedom we increase the decoupling of the geometry (cell, vertices) from the space of basis functions. We will associate different sets of basis functions with a cell. In 1D, all cells are intervals, while in 2D we can have cells that are triangles with straight sides, or any polygon, or in fact any two-dimensional geometry. Triangles and quadrilaterals are most common, though. The popular cell types in 3D are tetrahedra and hexahedra.

6.2 Extended finite element concept

The concept of a *finite element* is now

- a *reference cell* in a local reference coordinate system;
- a set of *basis functions* $\tilde{\varphi}_i$ defined on the cell;
- a set of *degrees of freedom* that uniquely determines the basis functions such that $\tilde{\varphi}_i = 1$ for degree of freedom number i and $\tilde{\varphi}_i = 0$ for all other degrees of freedom;

- a mapping between local and global degree of freedom numbers, here called the *dof map*;
- a geometric *mapping* of the reference cell onto to cell in the physical domain.

There must be a geometric description of a cell. This is trivial in 1D since the cell is an interval and is described by the interval limits, here called vertices. If the cell is $\Omega^{(e)} = [x_L, x_R]$, vertex 0 is x_L and vertex 1 is x_R. The reference cell in 1D is $[-1, 1]$ in the reference coordinate system X.

The expansion of u over one cell is often used:

$$u(x) = \tilde{u}(X) = \sum_r c_r \tilde{\varphi}_r(X), \quad x \in \Omega^{(e)},\ X \in [-1, 1], \tag{92}$$

where the sum is taken over the numbers of the degrees of freedom and c_r is the value of u for degree of freedom number r.

Our previous P1, P2, etc., elements are defined by introducing $d+1$ equally spaced nodes in the reference cell and saying that the degrees of freedom are the $d+1$ function values at these nodes. The basis functions must be 1 at one node and 0 at the others, and the Lagrange polynomials have exactly this property. The nodes can be numbered from left to right with associated degrees of freedom that are numbered in the same way. The degree of freedom mapping becomes what was previously represented by the `elements` lists. The cell mapping is the same affine mapping (61) as before.

6.3 Implementation

Implementationwise,

- we replace `nodes` by `vertices`;
- we introduce `cells` such that `cell[e][r]` gives the mapping from local vertex `r` in cell `e` to the global vertex number in `vertices`;
- we replace `elements` by `dof_map` (the contents are the same for Pd elements).

Consider the example from Section 3.1 where $\Omega = [0, 1]$ is divided into two cells, $\Omega^{(0)} = [0, 0.4]$ and $\Omega^{(1)} = [0.4, 1]$, as depicted in Figure 16. The vertices are $[0, 0.4, 1]$. Local vertex 0 and 1 are 0 and 0.4 in cell 0 and 0.4 and 1 in cell 1. A P2 element means that the degrees of freedom are the value of u at three equally spaced points (nodes) in each cell. The data structures become

```
vertices = [0, 0.4, 1]
cells = [[0, 1], [1, 2]]
dof_map = [[0, 1, 2], [2, 3, 4]]
```

If we would approximate f by piecewise constants, known as P0 elements, we simply introduce one point or node in an element, preferably $X = 0$, and define one degree of freedom, which is the function value at this node. Moreover, we set $\tilde{\varphi}_0(X) = 1$. The `cells` and `vertices` arrays remain the same, but `dof_map` is altered:

```
dof_map = [[0], [1]]
```

We use the `cells` and `vertices` lists to retrieve information on the geometry of a cell, while `dof_map` is the $q(e, r)$ mapping introduced earlier in the assembly of element matrices and vectors. For example, the `Omega_e` variable (representing the cell interval) in previous code snippets must now be computed as

```
Omega_e = [vertices[cells[e][0], vertices[cells[e][1]]
```

The assembly is done by

```
A[dof_map[e][r], dof_map[e][s]] += A_e[r,s]
b[dof_map[e][r]] += b_e[r]
```

We will hereafter drop the `nodes` and `elements` arrays and work exclusively with `cells`, `vertices`, and `dof_map`. The module `fe_approx1D_numint.py` now replaces the module `fe_approx1D` and offers similar functions that work with the new concepts:

```
from fe_approx1D_numint import *
x = sp.Symbol('x')
f = x*(1 - x)
N_e = 10
vertices, cells, dof_map = mesh_uniform(N_e, d=3, Omega=[0,1])
phi = [basis(len(dof_map[e])-1) for e in range(N_e)]
A, b = assemble(vertices, cells, dof_map, phi, f)
c = np.linalg.solve(A, b)
# Make very fine mesh and sample u(x) on this mesh for plotting
x_u, u = u_glob(c, vertices, cells, dof_map,
                resolution_per_element=51)
plot(x_u, u)
```

These steps are offered in the `approximate` function, which we here apply to see how well four P0 elements (piecewise constants) can approximate a parabola:

```
from fe_approx1D_numint import *
x=sp.Symbol("x")
for N_e in 4, 8:
    approximate(x*(1-x), d=0, N_e=N_e, Omega=[0,1])
```

Figure 32 shows the result.

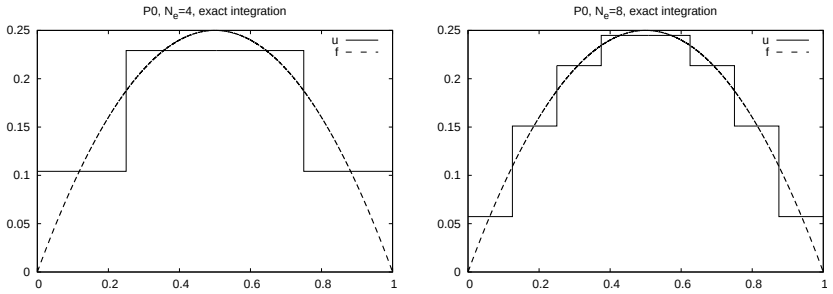

Figure 32: Approximation of a parabola by 4 (left) and 8 (right) P0 elements.

6.4 Computing the error of the approximation

So far we have focused on computing the coefficients c_j in the approximation $u(x) = \sum_j c_j \varphi_j$ as well as on plotting u and f for visual comparison. A more quantitative comparison needs to investigate the error $e(x) = f(x) - u(x)$. We mostly want a single number to reflect the error and use a norm for this purpose, usually the L^2 norm

$$||e||_{L^2} = \left(\int_\Omega e^2 dx \right)^{1/2}.$$

Since the finite element approximation is defined for all $x \in \Omega$, and we are interested in how $u(x)$ deviates from $f(x)$ through all the elements, we can either integrate analytically or use an accurate numerical approximation. The latter is more convenient as it is a generally feasible and simple approach. The idea is to sample $e(x)$ at a large number of points in each element. The function `u_glob` in the `fe_approx1D_numint` module does this for $u(x)$ and returns an array `x` with coordinates and an array `u` with the u values:

```
x, u = u_glob(c, vertices, cells, dof_map,
              resolution_per_element=101)
e = f(x) - u
```

Let us use the Trapezoidal method to approximate the integral. Because different elements may have different lengths, the `x` array has a non-uniformly distributed set of coordinates. Also, the `u_glob` function works in an element by element fashion such that coordinates at the boundaries between elements appear twice. We therefore need to use a "raw" version of the Trapezoidal rule where we just add up all the trapezoids:

$$\int_\Omega g(x)dx \approx \sum_{j=0}^{n-1} \frac{1}{2}(g(x_j) + g(x_{j+1}))(x_{j+1} - x_j),$$

if $x_0, \ldots, x_n$ are all the coordinates in `x`. In vectorized Python code,

```
g_x = g(x)
integral = 0.5*np.sum((g_x[:-1] + g_x[1:])*(x[1:] - x[:-1]))
```

Computing the L^2 norm of the error, here named `E`, is now achieved by

```
e2 = e**2
E = np.sqrt(0.5*np.sum((e2[:-1] + e2[1:])*(x[1:] - x[:-1]))
```

How does the error depend on h and d?

Theory and experiments show that the least squares or projection/Galerkin method in combination with Pd elements of equal length h has an error

$$||e||_{L^2} = Ch^{d+1}, \tag{93}$$

where C is a constant depending on f, but not on h or d.

6.5 Example: Cubic Hermite polynomials

The finite elements considered so far represent u as piecewise polynomials with discontinuous derivatives at the cell boundaries. Sometimes it is desirable to have continuous derivatives. A primary examples is the solution of differential equations with fourth-order derivatives where standard finite element formulations lead to a need for basis functions with continuous first-order derivatives. The most common type of such basis functions in 1D is the so-called cubic Hermite polynomials. The construction of such polynomials, as explained next, will further exemplify the concepts of a cell, vertex, degree of freedom, and dof map.

Given a reference cell $[-1, 1]$, we seek cubic polynomials with the values of the *function* and its *first-order derivative* at $X = -1$ and $X = 1$ as the four degrees of freedom. Let us number the degrees of freedom as

- 0: value of function at $X = -1$
- 1: value of first derivative at $X = -1$
- 2: value of function at $X = 1$
- 3: value of first derivative at $X = 1$

By having the derivatives as unknowns, we ensure that the derivative of a basis function in two neighboring elements is the same at the node points.

The four basis functions can be written in a general form

$$\tilde{\varphi}_i(X) = \sum_{j=0}^{3} C_{i,j} X^j,$$

with four coefficients $C_{i,j}$, $j = 0, 1, 2, 3$, to be determined for each i. The constraints that basis function number i must be 1 for degree of freedom number i and zero for the other three degrees of freedom, gives four equations to determine $C_{i,j}$ for each i. In mathematical detail,

$$\begin{aligned}
\tilde{\varphi}_0(-1) = 1, &\quad \tilde{\varphi}_0(1) = \tilde{\varphi}_0'(-1) = \tilde{\varphi}_i'(1) = 0, \\
\tilde{\varphi}_1'(-1) = 1, &\quad \tilde{\varphi}_1(-1) = \tilde{\varphi}_1(1) = \tilde{\varphi}_1'(1) = 0, \\
\tilde{\varphi}_2(1) = 1, &\quad \tilde{\varphi}_2(-1) = \tilde{\varphi}_2'(-1) = \tilde{\varphi}_2'(1) = 0, \\
\tilde{\varphi}_3'(1) = 1, &\quad \tilde{\varphi}_3(-1) = \tilde{\varphi}_3'(-1) = \tilde{\varphi}_3(1) = 0\,.
\end{aligned}$$

These four 4×4 linear equations can be solved, yielding the following formulas for the cubic basis functions:

$$\tilde{\varphi}_0(X) = 1 - \frac{3}{4}(X+1)^2 + \frac{1}{4}(X+1)^3 \tag{94}$$

$$\tilde{\varphi}_1(X) = -(X+1)(1 - \frac{1}{2}(X+1))^2 \tag{95}$$

$$\tilde{\varphi}_2(X) = \frac{3}{4}(X+1)^2 - \frac{1}{2}(X+1)^3 \tag{96}$$

$$\tilde{\varphi}_3(X) = -\frac{1}{2}(X+1)(\frac{1}{2}(X+1)^2 - (X+1)) \tag{97}$$

$$\tag{98}$$

The construction of the dof map needs a scheme for numbering the global degrees of freedom. A natural left-to-right numbering has the function value at vertex x_i as degree of freedom number $2i$ and the value of the derivative at x_i as degree of freedom number $2i+1$, $i = 0, \ldots, N_e + 1$.

7 Numerical integration

Finite element codes usually apply numerical approximations to integrals. Since the integrands in the coefficient matrix often are (lower-order) polynomials, integration rules that can integrate polynomials exactly are popular.

The numerical integration rules can be expressed in a common form,

$$\int_{-1}^{1} g(X)dX \approx \sum_{j=0}^{M} w_j g(\bar{X}_j), \tag{99}$$

where $\bar{X}_j$ are *integration points* and w_j are *integration weights*, $j = 0, \ldots, M$. Different rules correspond to different choices of points and weights.

The very simplest method is the *Midpoint rule*,

$$\int_{-1}^{1} g(X)dX \approx 2g(0), \quad \bar{X}_0 = 0,\ w_0 = 2, \tag{100}$$

which integrates linear functions exactly.

7.1 Newton-Cotes rules

The Newton-Cotes[8] rules are based on a fixed uniform distribution of the integration points. The first two formulas in this family are the well-known *Trapezoidal rule*,

$$\int_{-1}^{1} g(X)dX \approx g(-1) + g(1), \quad \bar{X}_0 = -1,\ \bar{X}_1 = 1,\ w_0 = w_1 = 1, \tag{101}$$

and *Simpson's rule*,

$$\int_{-1}^{1} g(X)dX \approx \frac{1}{3}\left(g(-1) + 4g(0) + g(1)\right), \tag{102}$$

where

$$\bar{X}_0 = -1,\ \bar{X}_1 = 0,\ \bar{X}_2 = 1,\ w_0 = w_2 = \frac{1}{3},\ w_1 = \frac{4}{3}. \tag{103}$$

Newton-Cotes rules up to five points is supported in the module file `numint.py`[9].

For higher accuracy one can divide the reference cell into a set of subintervals and use the rules above on each subinterval. This approach results in *composite* rules, well-known from basic introductions to numerical integration of $\int_a^b f(x)dx$.

7.2 Gauss-Legendre rules with optimized points

More accurate rules, for a given M, arise if the location of the integration points are optimized for polynomial integrands. The Gauss-Legendre rules[10] (also known as Gauss-Legendre quadrature or Gaussian quadrature) constitute one such class of integration methods. Two widely applied Gauss-Legendre rules in this family have the choice

$$M = 1: \quad \bar{X}_0 = -\frac{1}{\sqrt{3}},\ \bar{X}_1 = \frac{1}{\sqrt{3}},\ w_0 = w_1 = 1 \tag{104}$$

$$M = 2: \quad \bar{X}_0 = -\sqrt{\frac{3}{5}},\ \bar{X}_0 = 0,\ \bar{X}_2 = \sqrt{\frac{3}{5}},\ w_0 = w_2 = \frac{5}{9},\ w_1 = \frac{8}{9}. \tag{105}$$

These rules integrate 3rd and 5th degree polynomials exactly. In general, an M-point Gauss-Legendre rule integrates a polynomial of degree $2M + 1$ exactly. The code `numint.py` contains a large collection of Gauss-Legendre rules.

[8] `http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas`
[9] `http://tinyurl.com/nm5587k/fem/numint.py`
[10] `http://en.wikipedia.org/wiki/Gaussian_quadrature`

8 Approximation of functions in 2D

All the concepts and algorithms developed for approximation of 1D functions $f(x)$ can readily be extended to 2D functions $f(x,y)$ and 3D functions $f(x,y,z)$. Basically, the extensions consists of defining basis functions $\psi_i(x,y)$ or $\psi_i(x,y,z)$ over some domain Ω, and for the least squares and Galerkin methods, the integration is done over Ω.

As in 1D, the least squares and projection/Galerkin methods two lead to linear systems

$$\sum_{j\in\mathcal{I}_s} A_{i,j}c_j = b_i, \quad i \in \mathcal{I}_s,$$
$$A_{i,j} = (\psi_i, \psi_j),$$
$$b_i = (f, \psi_i),$$

where the inner product of two functions $f(x,y)$ and $g(x,y)$ is defined completely analogously to the 1D case (24):

$$(f,g) = \int_\Omega f(x,y)g(x,y)dxdy \tag{106}$$

8.1 2D basis functions as tensor products of 1D functions

One straightforward way to construct a basis in 2D is to combine 1D basis functions. Say we have the 1D vector space

$$V_x = \text{span}\{\hat{\psi}_0(x), \ldots, \hat{\psi}_{N_x}(x)\}\,. \tag{107}$$

A similar space for variation in y can be defined,

$$V_y = \text{span}\{\hat{\psi}_0(y), \ldots, \hat{\psi}_{N_y}(y)\}\,. \tag{108}$$

We can then form 2D basis functions as *tensor products* of 1D basis functions.

Tensor products.

Given two vectors $a = (a_0, \ldots, a_M)$ and $b = (b_0, \ldots, b_N)$, their *outer tensor product*, also called the *dyadic product*, is $p = a \otimes b$, defined through

$$p_{i,j} = a_i b_j, \quad i = 0, \ldots, M,\ j = 0, \ldots, N\,.$$

In the tensor terminology, a and b are first-order tensors (vectors with one index, also termed rank-1 tensors), and then their outer tensor product is a second-order tensor (matrix with two indices, also termed rank-2 tensor). The corresponding *inner tensor product* is the well-known scalar or dot product of two vectors: $p = a \cdot b = \sum_{j=0}^{N} a_j b_j$. Now, p is a rank-0 tensor.

Tensors are typically represented by arrays in computer code. In the above example, a and b are represented by one-dimensional arrays of length M and N, respectively, while $p = a \otimes b$ must be represented by a two-dimensional array of size $M \times N$.

Tensor products[a] can be used in a variety of context.

[a] http://en.wikipedia.org/wiki/Tensor_product

Given the vector spaces V_x and V_y as defined in (107) and (108), the tensor product space $V = V_x \otimes V_y$ has a basis formed as the tensor product of the basis for V_x and V_y. That is, if $\{\varphi_i(x)\}_{i\in\mathcal{I}_x}$ and $\{\varphi_i(y)\}_{i\in\mathcal{I}_y}$ are basis for V_x and V_y, respectively, the elements in the basis for V arise from the tensor product: $\{\varphi_i(x)\varphi_j(y)\}_{i\in\mathcal{I}_x,j\in\mathcal{I}_y}$. The index sets are $I_x = \{0,\ldots,N_x\}$ and $I_y = \{0,\ldots,N_y\}$.

The notation for a basis function in 2D can employ a double index as in

$$\psi_{p,q}(x,y) = \hat{\psi}_p(x)\hat{\psi}_q(y), \quad p \in \mathcal{I}_x, q \in \mathcal{I}_y\,.$$

The expansion for u is then written as a double sum

$$u = \sum_{p\in\mathcal{I}_x}\sum_{q\in\mathcal{I}_y} c_{p,q}\psi_{p,q}(x,y)\,.$$

Alternatively, we may employ a single index,

$$\psi_i(x,y) = \hat{\psi}_p(x)\hat{\psi}_q(y),$$

and use the standard form for u,

$$u = \sum_{j\in\mathcal{I}_s} c_j\psi_j(x,y)\,.$$

The single index is related to the double index through $i = p(N_y+1)+q$ or $i = q(N_x+1)+p$.

8.2 Example: Polynomial basis in 2D

Suppose we choose $\hat{\psi}_p(x) = x^p$, and try an approximation with $N_x = N_y = 1$:

$$\psi_{0,0} = 1, \quad \psi_{1,0} = x, \quad \psi_{0,1} = y, \quad \psi_{1,1} = xy\,.$$

Using a mapping to one index like $i = q(N_x+1)+p$, we get

$$\psi_0 = 1, \quad \psi_1 = x, \quad \psi_2 = y, \quad \psi_3 = xy\,.$$

With the specific choice $f(x,y) = (1+x^2)(1+2y^2)$ on $\Omega = [0,L_x]\times[0,L_y]$, we can perform actual calculations:

$$A_{0,0} = (\psi_0, \psi_0) = \int_0^{L_y} \int_0^{L_x} \psi_0(x,y)^2 dxdy = \int_0^{L_y} \int_0^{L_x} dxdy = L_x L_y,$$
$$A_{1,0} = (\psi_1, \psi_0) = \int_0^{L_y} \int_0^{L_x} x dxdy = \frac{1}{2} L_x^2 L_y,$$
$$A_{0,1} = (\psi_0, \psi_1) = \int_0^{L_y} \int_0^{L_x} y dxdy = \frac{1}{2} L_y^2 L_x,$$
$$A_{0,1} = (\psi_0, \psi_1) = \int_0^{L_y} \int_0^{L_x} xy dxdy = \int_0^{L_y} y dy \int_0^{L_x} x dx = \frac{1}{4} L_y^2 L_x^2 .$$

The right-hand side vector has the entries

$$\begin{aligned}
b_0 &= (\psi_0, f) = \int_0^{L_y} \int_0^{L_x} 1 \cdot (1+x^2)(1+2y^2) dxdy \\
&= \int_0^{L_y} (1+2y^2) dy \int_0^{L_x} (1+x^2) dx = (L_y + \frac{2}{3} L_y^3)(L_x + \frac{1}{3} L_x^3) \\
b_1 &= (\psi_1, f) = \int_0^{L_y} \int_0^{L_x} x(1+x^2)(1+2y^2) dxdy \\
&= \int_0^{L_y} (1+2y^2) dy \int_0^{L_x} x(1+x^2) dx = (L_y + \frac{2}{3} L_y^3)(\frac{1}{2} L_x^2 + \frac{1}{4} L_x^4) \\
b_2 &= (\psi_2, f) = \int_0^{L_y} \int_0^{L_x} y(1+x^2)(1+2y^2) dxdy \\
&= \int_0^{L_y} y(1+2y^2) dy \int_0^{L_x} (1+x^2) dx = (\frac{1}{2} L_y + \frac{1}{2} L_y^4)(L_x + \frac{1}{3} L_x^3) \\
b_3 &= (\psi_2, f) = \int_0^{L_y} \int_0^{L_x} xy(1+x^2)(1+2y^2) dxdy \\
&= \int_0^{L_y} y(1+2y^2) dy \int_0^{L_x} x(1+x^2) dx = (\frac{1}{2} L_y^2 + \frac{1}{2} L_y^4)(\frac{1}{2} L_x^2 + \frac{1}{4} L_x^4) .
\end{aligned}$$

There is a general pattern in these calculations that we can explore. An arbitrary matrix entry has the formula

$$\begin{aligned}
A_{i,j} &= (\psi_i, \psi_j) = \int_0^{L_y} \int_0^{L_x} \psi_i \psi_j dxdy \\
&= \int_0^{L_y} \int_0^{L_x} \psi_{p,q} \psi_{r,s} dxdy = \int_0^{L_y} \int_0^{L_x} \hat{\psi}_p(x) \hat{\psi}_q(y) \hat{\psi}_r(x) \hat{\psi}_s(y) dxdy \\
&= \int_0^{L_y} \hat{\psi}_q(y) \hat{\psi}_s(y) dy \int_0^{L_x} \hat{\psi}_p(x) \hat{\psi}_r(x) dx \\
&= \hat{A}^{(x)}_{p,r} \hat{A}^{(y)}_{q,s},
\end{aligned}$$

where

$$\hat{A}^{(x)}_{p,r} = \int_0^{L_x} \hat{\psi}_p(x)\hat{\psi}_r(x)dx, \quad \hat{A}^{(y)}_{q,s} = \int_0^{L_y} \hat{\psi}_q(y)\hat{\psi}_s(y)dy,$$

are matrix entries for one-dimensional approximations. Moreover, $i = qN_y + q$ and $j = sN_y + r$.

With $\hat{\psi}_p(x) = x^p$ we have

$$\hat{A}^{(x)}_{p,r} = \frac{1}{p+r+1}L_x^{p+r+1}, \quad \hat{A}^{(y)}_{q,s} = \frac{1}{q+s+1}L_y^{q+s+1},$$

and

$$A_{i,j} = \hat{A}^{(x)}_{p,r}\hat{A}^{(y)}_{q,s} = \frac{1}{p+r+1}L_x^{p+r+1}\frac{1}{q+s+1}L_y^{q+s+1},$$

for $p, r \in \mathcal{I}_x$ and $q, s \in \mathcal{I}_y$.

Corresponding reasoning for the right-hand side leads to

$$\begin{aligned}
b_i = (\psi_i, f) &= \int_0^{L_y}\int_0^{L_x} \psi_i f\, dxdx \\
&= \int_0^{L_y}\int_0^{L_x} \hat{\psi}_p(x)\hat{\psi}_q(y) f\, dxdx \\
&= \int_0^{L_y} \hat{\psi}_q(y)(1+2y^2)dy \int_0^{L_y} \hat{\psi}_p(x)x^p(1+x^2)dx \\
&= \int_0^{L_y} y^q(1+2y^2)dy \int_0^{L_y} x^p(1+x^2)dx \\
&= (\frac{1}{q+1}L_y^{q+1} + \frac{2}{q+3}L_y^{q+3})(\frac{1}{p+1}L_x^{p+1} + \frac{2}{q+3}L_x^{p+3})
\end{aligned}$$

Choosing $L_x = L_y = 2$, we have

$$A = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & \frac{16}{3} & 4 & \frac{16}{3} \\ 4 & 4 & \frac{16}{3} & \frac{16}{3} \\ 4 & \frac{16}{3} & \frac{16}{3} & \frac{64}{9} \end{bmatrix}, \quad b = \begin{bmatrix} \frac{308}{9} \\ \frac{140}{3} \\ 44 \\ 60 \end{bmatrix}, \quad c = \begin{bmatrix} -\frac{1}{9} \\ \frac{4}{3} \\ -\frac{2}{3} \\ 8 \end{bmatrix}.$$

Figure 33 illustrates the result.

8.3 Implementation

The `least_squares` function from Section 2.8 and/or the file `approx1D.py`[11] can with very small modifications solve 2D approximation problems. First, let `Omega` now be a list of the intervals in x and y direction. For example, $\Omega = [0, L_x] \times [0, L_y]$ can be represented by `Omega = [[0, L_x], [0, L_y]]`.

Second, the symbolic integration must be extended to 2D:

[11] `http://tinyurl.com/nm5587k/fem/fe_approx1D.py`

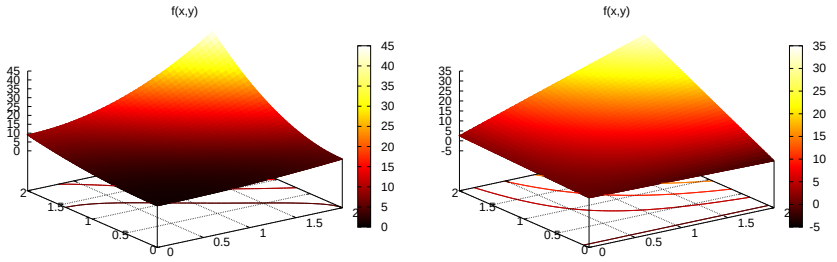

Figure 33: Approximation of a 2D quadratic function (left) by a 2D bilinear function (right) using the Galerkin or least squares method.

```
import sympy as sp

integrand = psi[i]*psi[j]
I = sp.integrate(integrand,
                 (x, Omega[0][0], Omega[0][1]),
                 (y, Omega[1][0], Omega[1][1]))
```

provided `integrand` is an expression involving the `sympy` symbols `x` and `y`. The 2D version of numerical integration becomes

```
if isinstance(I, sp.Integral):
    integrand = sp.lambdify([x,y], integrand)
    I = sp.mpmath.quad(integrand,
                       [Omega[0][0], Omega[0][1]],
                       [Omega[1][0], Omega[1][1]])
```

The right-hand side integrals are modified in a similar way.

Third, we must construct a list of 2D basis functions. Here are two examples based on tensor products of 1D "Taylor-style" polynomials x^i and 1D sine functions $\sin((i+1)\pi x)$:

```
def taylor(x, y, Nx, Ny):
    return [x**i*y**j for i in range(Nx+1) for j in range(Ny+1)]

def sines(x, y, Nx, Ny):
    return [sp.sin(sp.pi*(i+1)*x)*sp.sin(sp.pi*(j+1)*y)
            for i in range(Nx+1) for j in range(Ny+1)]
```

The complete code appears in `approx2D.py`[12].

The previous hand calculation where a quadratic f was approximated by a bilinear function can be computed symbolically by

```
>>> from approx2D import *
>>> f = (1+x**2)*(1+2*y**2)
>>> psi = taylor(x, y, 1, 1)
>>> Omega = [[0, 2], [0, 2]]
```

[12] `http://tinyurl.com/nm5587k/fem/fe_approx2D.py`

```
>>> u, c = least_squares(f, psi, Omega)
>>> print u
8*x*y - 2*x/3 + 4*y/3 - 1/9
>>> print sp.expand(f)
2*x**2*y**2 + x**2 + 2*y**2 + 1
```

We may continue with adding higher powers to the basis:

```
>>> psi = taylor(x, y, 2, 2)
>>> u, c = least_squares(f, psi, Omega)
>>> print u
2*x**2*y**2 + x**2 + 2*y**2 + 1
>>> print u-f
0
```

For $N_x \geq 2$ and $N_y \geq 2$ we recover the exact function f, as expected, since in that case $f \in V$ (see Section 2.5).

8.4 Extension to 3D

Extension to 3D is in principle straightforward once the 2D extension is understood. The only major difference is that we need the repeated outer tensor product,

$$V = V_x \otimes V_y \otimes V_z\,.$$

In general, given vectors (first-order tensors) $a^{(q)} = (a_0^{(q)}, \ldots, a_{N_q}^{(q)}$, $q = 0, \ldots, m$, the tensor product $p = a^{(0)} \otimes \cdots \otimes a^m$ has elements

$$p_{i_0,i_1,\ldots,i_m} = a_{i_1}^{(0)} a_{i_1}^{(1)} \cdots a_{i_m}^{(m)}\,.$$

The basis functions in 3D are then

$$\psi_{p,q,r}(x,y,z) = \hat{\psi}_p(x)\hat{\psi}_q(y)\hat{\psi}_r(z),$$

with $p \in \mathcal{I}_x$, $q \in \mathcal{I}_y$, $r \in \mathcal{I}_z$. The expansion of u becomes

$$u(x,y,z) = \sum_{p\in\mathcal{I}_x}\sum_{q\in\mathcal{I}_y}\sum_{r\in\mathcal{I}_z} c_{p,q,r}\psi_{p,q,r}(x,y,z)\,.$$

A single index can be introduced also here, e.g., $i = N_x N_y r + q_N x + p$, $u = \sum_i c_i\psi_i(x,y,z)$.

Use of tensor product spaces.

Constructing a multi-dimensional space and basis from tensor products of 1D spaces is a standard technique when working with global basis functions. In the world of finite elements, constructing basis functions by tensor products is much used on quadrilateral and hexahedra cell shapes, but not on triangles and tetrahedra. Also, the global finite element basis functions

are almost exclusively denoted by a single index and not by the natural tuple of indices that arises from tensor products.

9 Finite elements in 2D and 3D

Finite element approximation is particularly powerful in 2D and 3D because the method can handle a geometrically complex domain Ω with ease. The principal idea is, as in 1D, to divide the domain into cells and use polynomials for approximating a function over a cell. Two popular cell shapes are triangles and the quadrilaterals. Figures 34, 35, and 36 provide examples. P1 elements means linear functions ($a_0 + a_1x + a_2y$) over triangles, while Q1 elements have bilinear functions ($a_0 + a_1x + a_2y + a_3xy$) over rectangular cells. Higher-order elements can easily be defined.

9.1 Basis functions over triangles in the physical domain

Cells with triangular shape will be in main focus here. With the P1 triangular element, u is a linear function over each cell, as depicted in Figure 37, with discontinuous derivatives at the cell boundaries.

We give the vertices of the cells global and local numbers as in 1D. The degrees of freedom in the P1 element are the function values at a set of nodes, which are the three vertices. The basis function $\varphi_i(x, y)$ is then 1 at the vertex with global vertex number i and zero at all other vertices. On an element, the three degrees of freedom uniquely determine the linear basis functions in that element, as usual. The global $\varphi_i(x, y)$ function is then a combination of the linear functions (planar surfaces) over all the neighboring cells that have vertex number i in common. Figure 38 tries to illustrate the shape of such a "pyramid"-like function.

Element matrices and vectors. As in 1D, we split the integral over Ω into a sum of integrals over cells. Also as in 1D, φ_i overlaps φ_j (i.e., $\varphi_i\varphi_j \neq 0$) if and only if i and j are vertices in the same cell. Therefore, the integral of $\varphi_i\varphi_j$ over an element is nonzero only when i and j run over the vertex numbers in the element. These nonzero contributions to the coefficient matrix are, as in 1D, collected in an element matrix. The size of the element matrix becomes 3×3 since there are three degrees of freedom that i and j run over. Again, as in 1D, we number the local vertices in a cell, starting at 0, and add the entries in the element matrix into the global system matrix, exactly as in 1D. All details and code appear below.

9.2 Basis functions over triangles in the reference cell

As in 1D, we can define the basis functions and the degrees of freedom in a reference cell and then use a mapping from the reference coordinate system to

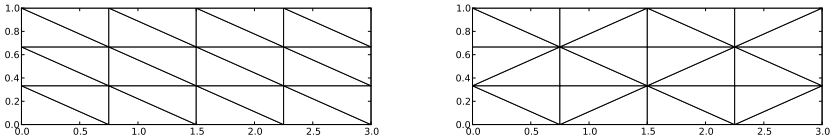

Figure 34: Examples on 2D P1 elements.

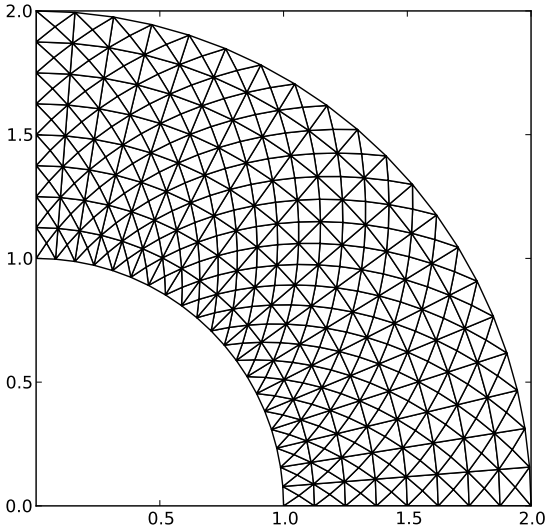

Figure 35: Examples on 2D P1 elements in a deformed geometry.

the physical coordinate system. We also have a mapping of local degrees of freedom numbers to global degrees of freedom numbers.

The reference cell in an (X, Y) coordinate system has vertices $(0, 0)$, $(1, 0)$, and $(0, 1)$, corresponding to local vertex numbers 0, 1, and 2, respectively. The P1 element has linear functions $\tilde{\varphi}_r(X, Y)$ as basis functions, $r = 0, 1, 2$. Since a linear function $\tilde{\varphi}_r(X, Y)$ in 2D is on the form $C_{r,0} + C_{r,1}X + C_{r,2}Y$, and hence has three parameters $C_{r,0}$, $C_{r,1}$, and $C_{r,2}$, we need three degrees of freedom. These are in general taken as the function values at a set of nodes. For the P1 element the set of nodes is the three vertices. Figure 39 displays the geometry of the element and the location of the nodes.

Requiring $\tilde{\varphi}_r = 1$ at node number r and $\tilde{\varphi}_r = 0$ at the two other nodes, gives three linear equations to determine $C_{r,0}$, $C_{r,1}$, and $C_{r,2}$. The result is

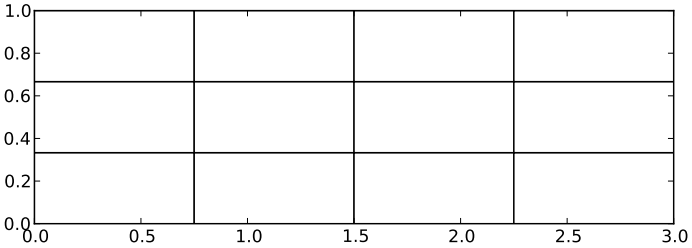

Figure 36: Examples on 2D Q1 elements.

$$\tilde{\varphi}_0(X,Y) = 1 - X - Y, \tag{109}$$
$$\tilde{\varphi}_1(X,Y) = X, \tag{110}$$
$$\tilde{\varphi}_2(X,Y) = Y \tag{111}$$

Higher-order approximations are obtained by increasing the polynomial order, adding additional nodes, and letting the degrees of freedom be function values at the nodes. Figure 40 shows the location of the six nodes in the P2 element.

A polynomial of degree p in X and Y has $n_p = (p+1)(p+2)/2$ terms and hence needs n_p nodes. The values at the nodes constitute n_p degrees of freedom. The location of the nodes for $\tilde{\varphi}_r$ up to degree 6 is displayed in Figure 41.

The generalization to 3D is straightforward: the reference element is a tetrahedron[13] with vertices $(0,0,0)$, $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ in a X,Y,Z reference coordinate system. The P1 element has its degrees of freedom as four nodes, which are the four vertices, see Figure 42. The P2 element adds additional nodes along the edges of the cell, yielding a total of 10 nodes and degrees of freedom, see Figure 43.

The interval in 1D, the triangle in 2D, the tetrahedron in 3D, and its generalizations to higher space dimensions are known as *simplex* cells (the geometry) or *simplex* elements (the geometry, basis functions, degrees of freedom,

[13] `http://en.wikipedia.org/wiki/Tetrahedron`

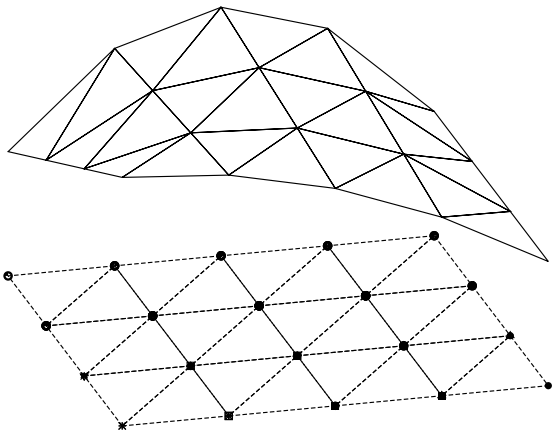

Figure 37: Example on piecewise linear 2D functions defined on triangles.

etc.). The plural forms simplices[14] and simplexes are also a much used shorter terms when referring to this type of cells or elements. The side of a simplex is called a *face*, while the tetrahedron also has *edges*.

Acknowledgment. Figures 39-43 are created by Anders Logg and taken from the FEniCS book[15]: *Automated Solution of Differential Equations by the Finite Element Method*, edited by A. Logg, K.-A. Mardal, and G. N. Wells, published by Springer[16], 2012.

9.3 Affine mapping of the reference cell

Let $\tilde{\varphi}_r^{(1)}$ denote the basis functions associated with the P1 element in 1D, 2D, or 3D, and let $\boldsymbol{x}_{q(e,r)}$ be the physical coordinates of local vertex number r in cell e. Furthermore, let $\boldsymbol{X}$ be a point in the reference coordinate system corresponding to the point $\boldsymbol{x}$ in the physical coordinate system. The affine mapping of any $\boldsymbol{X}$ onto $\boldsymbol{x}$ is then defined by

$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r^{(1)}(\boldsymbol{X})\boldsymbol{x}_{q(e,r)}, \tag{112}$$

where r runs over the local vertex numbers in the cell. The affine mapping essentially stretches, translates, and rotates the triangle. Straight or planar

[14] `http://en.wikipedia.org/wiki/Simplex`
[15] `https://launchpad.net/fenics-book`
[16] `http://goo.gl/lbyVMH`

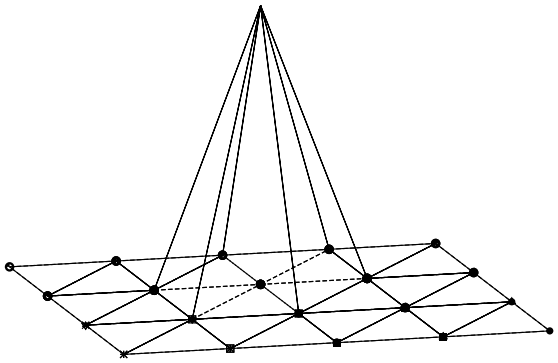

Figure 38: Example on a piecewise linear 2D basis function over a patch of triangles.

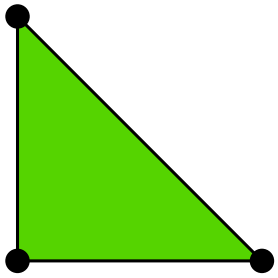

Figure 39: 2D P1 element.

faces of the reference cell are therefore mapped onto straight or planar faces in the physical coordinate system. The mapping can be used for both P1 and higher-order elements, but note that the mapping itself always applies the P1 basis functions.

9.4 Isoparametric mapping of the reference cell

Instead of using the P1 basis functions in the mapping (112), we may use the basis functions of the actual Pd element:

$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r(\boldsymbol{X})\boldsymbol{x}_{q(e,r)}, \tag{113}$$

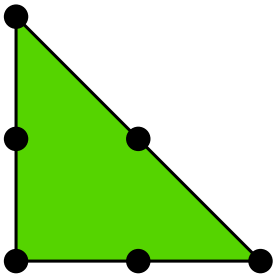

Figure 40: 2D P2 element.

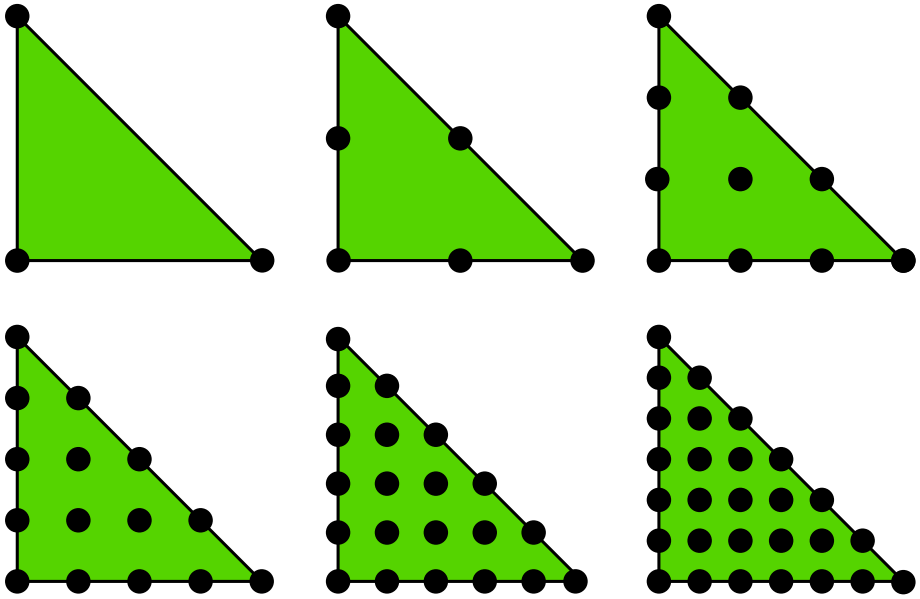

Figure 41: 2D P1, P2, P3, P4, P5, and P6 elements.

where r runs over all nodes, i.e., all points associated with the degrees of freedom. This is called an *isoparametric mapping*. For P1 elements it is identical to the affine mapping (112), but for higher-order elements the mapping of the straight or planar faces of the reference cell will result in a *curved* face in the physical coordinate system. For example, when we use the basis functions of the triangular P2 element in 2D in (113), the straight faces of the reference triangle are mapped onto curved faces of parabolic shape in the physical coordinate system, see Figure 45.

From (112) or (113) it is easy to realize that the vertices are correctly mapped. Consider a vertex with local number s. Then $\tilde{\varphi}_s = 1$ at this vertex and zero at the others. This means that only one term in the sum is nonzero and $\boldsymbol{x} = \boldsymbol{x}_{q(e,s)}$, which is the coordinate of this vertex in the global coordinate system.

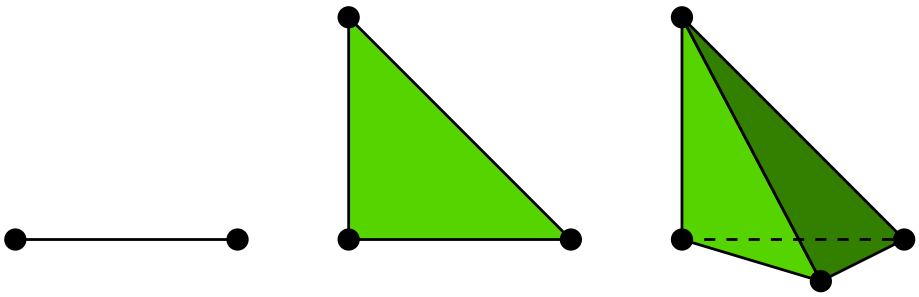

Figure 42: P1 elements in 1D, 2D, and 3D.

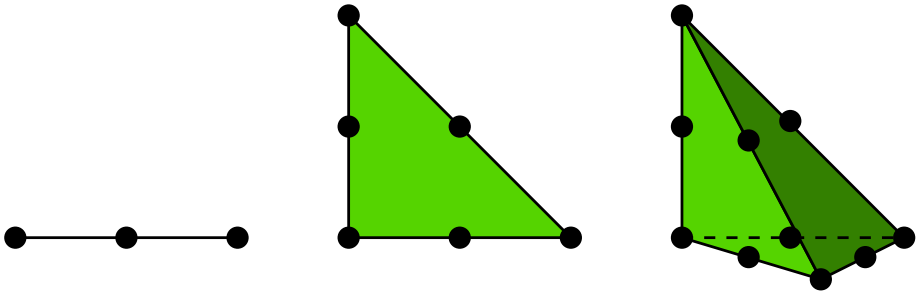

Figure 43: P2 elements in 1D, 2D, and 3D.

9.5 Computing integrals

Let $\tilde{\Omega}^r$ denote the reference cell and $\Omega^{(e)}$ the cell in the physical coordinate system. The transformation of the integral from the physical to the reference coordinate system reads

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})\varphi_j(\boldsymbol{x})\,\mathrm{d}x = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\boldsymbol{X})\tilde{\varphi}_j(\boldsymbol{X})\det J\;\mathrm{d}X, \tag{114}$$

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})f(\boldsymbol{x})\,\mathrm{d}x = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\boldsymbol{X})f(\boldsymbol{x}(\boldsymbol{X}))\det J\;\mathrm{d}X, \tag{115}$$

where $\mathrm{d}x$ means the infinitesimal area element $dxdy$ in 2D and $dxdydz$ in 3D, with a similar definition of $\mathrm{d}X$. The quantity $\det J$ is the determinant of the Jacobian of the mapping $\boldsymbol{x}(\boldsymbol{X})$. In 2D,

$$J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial x}{\partial Y} \\ \frac{\partial y}{\partial X} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \det J = \frac{\partial x}{\partial X}\frac{\partial y}{\partial Y} - \frac{\partial x}{\partial Y}\frac{\partial y}{\partial X}. \tag{116}$$

With the affine mapping (112), $\det J = 2\Delta$, where Δ is the area or volume of the cell in the physical coordinate system.

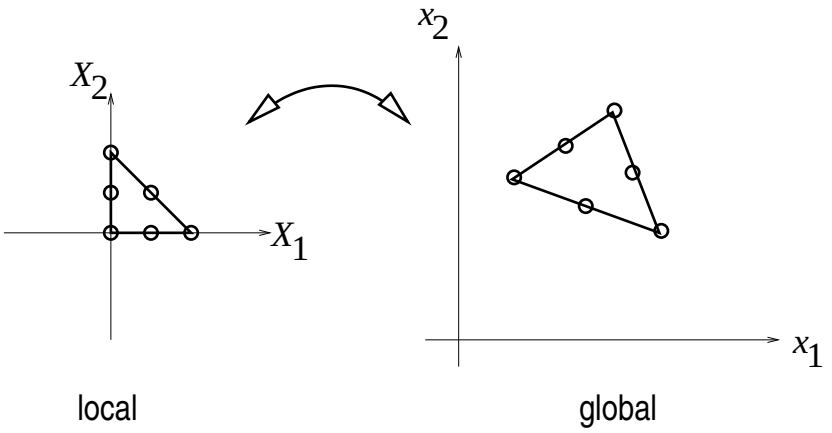

Figure 44: Affine mapping of a P1 element.

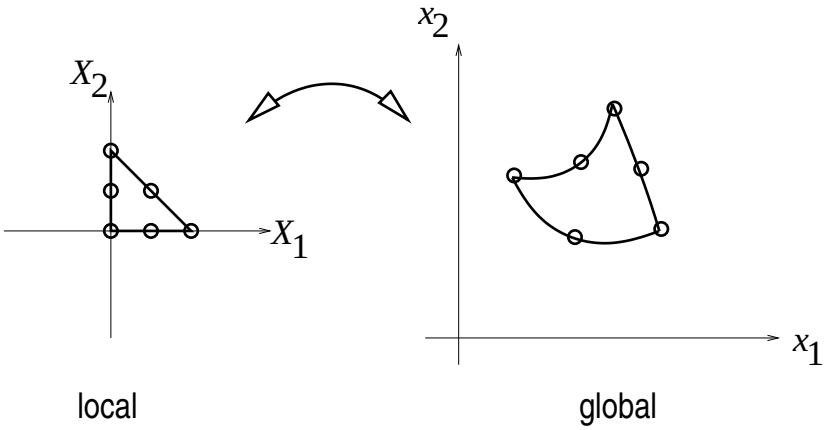

Figure 45: Isoparametric mapping of a P2 element.

Remark. Observe that finite elements in 2D and 3D builds on the same *ideas* and *concepts* as in 1D, but there is simply much more to compute because the specific mathematical formulas in 2D and 3D are more complicated and the book keeping with dof maps also gets more complicated. The manual work is tedious, lengthy, and error-prone so automation by the computer is a must.

10 Exercises

Exercise 1: Linear algebra refresher I

Look up the topic of *vector space* in your favorite linear algebra book or search for the term at Wikipedia. Prove that vectors in the plane (a, b) form a vector

space by showing that all the axioms of a vector space are satisfied. Similarly, prove that all linear functions of the form $ax + b$ constitute a vector space, $a, b \in \mathbb{R}$.

On the contrary, show that all quadratic functions of the form $1 + ax^2 + bx$ *do not* constitute a vector space. Filename: `linalg1.pdf`.

Exercise 2: Linear algebra refresher II

As an extension of Exercise 1, check out the topic of *inner product spaces*. Suggest a possible inner product for the space of all linear functions of the form $ax + b$, $a, b \in \mathbb{R}$. Show that this inner product satisfies the general requirements of an inner product in a vector space. Filename: `linalg2.pdf`.

Exercise 3: Approximate a three-dimensional vector in a plane

Given $\boldsymbol{f} = (1, 1, 1)$ in $\mathbb{R}^3$, find the best approximation vector $\boldsymbol{u}$ in the plane spanned by the unit vectors $(1, 0)$ and $(0, 1)$. Repeat the calculations using the vectors $(2, 1)$ and $(1, 2)$. Filename: `vec111_approx.pdf`.

Exercise 4: Approximate the exponential function by power functions

Let V be a function space with basis functions x^i, $i = 0, 1, \ldots, N$. Find the best approximation to $f(x) = \exp(-x)$ on $\Omega = [0, 8]$ among all functions in V for $N = 2, 4, 6$. Illustrate the three approximations in three separate plots.

Hint. The exercise is easy to solve if you apply the `lest_squares` and `comparison_plot` functions in the `approx1D.py` module.
Filename: `exp_powers.py`.

Exercise 5: Approximate the sine function by power functions

In this exercise we want to approximate the sine function by polynomials of order $N + 1$. Consider two bases:

$$V_1 = \{x, x^3, x^5, \ldots, x^{N-2}, x^N\},$$
$$V_2 = \{1, x, x^2, x^3, \ldots, x^N\}.$$

The basis V_1 is motivated by the fact that the Taylor polynomial approximation to the sine function has only odd powers, while V_2 is motivated by the assumption that also the even powers could improve the approximation in a least-squares setting.

Compute the best approximation to $f(x) = \sin(x)$ among all functions in V_1 and V_2 on two domains of increasing sizes: $\Omega_{1,k} = [0, k\pi]$, $k = 2, 3 \ldots, 6$ and $\Omega_{2,k} = [-k\pi/2, k\pi/2]$, $k = 2, 3 \ldots, 6$. Make plots for all combinations of V_1, V_2, Ω_1, Ω_2, $k = 2, 3, \ldots, 6$.

Add a plot of the N-th degree Taylor polynomial approximation of $\sin(x)$ around $x = 0$.

Hint. You can make a loop over V_1 and V_2, a loop over Ω_1 and Ω_2, and a loop over k. Inside the loops, call the functions `least_squares` and `comparison_plot` from the `approx1D` module. $N = 9$ is a suggested value.
Filename: `sin_powers.py`.

Exercise 6: Approximate a steep function by sines

Find the best approximation of $f(x) = \tanh(s(x - \pi))$ on $[0, 2\pi]$ in the space V with basis $\psi_i(x) = \sin((2i + 1)x)$, $i \in \mathcal{I}_s = \{0, \ldots, N\}$. Make a movie showing how $u = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$ approximates $f(x)$ as N grows. Choose s such that f is steep ($s = 20$ may be appropriate).

Hint. One may naively call the `least_squares_orth` and `comparison_plot` from the `approx1D` module in a loop and extend the basis with one new element in each pass. This approach implies a lot of recomputations. A more efficient strategy is to let `least_squares_orth` compute with only one basis function at a time and accumulate the corresponding `u` in the total solution.
Filename: `tanh_sines_approx.py`.

Remarks. Approximation of a discontinuous (or steep) $f(x)$ by sines, results in slow convergence and oscillatory behavior of the approximation close to the abrupt changes in f. This is known as the Gibb's phenomenon[17].

Exercise 7: Animate the approximation of a steep function by sines

Make a movie that shows how the approximation in Exercise 6 is improved as N grows. Illustrate a smooth case where $s = 0.5$ and a steep case where $s = 20$ in the $\tanh(s(x - \pi))$ function. Filename: `tanh_sines_approx_movie.py`.

Exercise 8: Fourier series as a least squares approximation

Given a function $f(x)$ on an interval $[0, L]$, look up the formula for the coefficients a_j and b_j in the Fourier series of f:

$$f(x) = a_0 + \sum_{j=1}^{\infty} a_j \cos\left(j\frac{\pi x}{L}\right) + \sum_{j=1}^{\infty} b_j \sin\left(j\frac{\pi x}{L}\right) .$$

[17] `http://en.wikipedia.org/wiki/Gibbs_phenomenon`

Let an infinite-dimensional vector space V have the basis functions $\cos j\frac{\pi x}{L}$ for $j = 0, 1, \ldots, \infty$ and $\sin j\frac{\pi x}{L}$ for $j = 1, \ldots, \infty$. Show that the least squares approximation method from Section 2 leads to a linear system whose solution coincides with the standard formulas for the coefficients in a Fourier series of $f(x)$ (see also Section 2.7). You may choose

$$\psi_{2i} = \cos\left(i\frac{\pi}{L}x\right), \quad \psi_{2i+1} = \sin\left(i\frac{\pi}{L}x\right),$$

for $i = 0, 1, \ldots, N \to \infty$.

Choose $f(x) = \tanh(s(x - \frac{1}{2}))$ on $\Omega = [0, 1]$, which is a smooth function, but with considerable steepness around $x = 1/2$ as s grows in size. Calculate the coefficients in the Fourier expansion by solving the linear system, arising from the least squares or Galerkin methods, by hand. Plot some truncated versions of the series together with $f(x)$ to show how the series expansion converges for $s = 10$ and $s = 100$. Filename: `Fourier_approx.py`.

Exercise 9: Approximate a steep function by Lagrange polynomials

Use interpolation/collocation with uniformly distributed points and Chebychev nodes to approximate

$$f(x) = -\tanh(s(x - \frac{1}{2})), \quad x \in [0, 1],$$

by Lagrange polynomials for $s = 10$ and $s = 100$, and $N = 3, 6, 9, 11$. Make separate plots of the approximation for each combination of s, point type (Chebyshev or uniform), and N. Filename: `tanh_Lagrange.py`.

Exercise 10: Define nodes and elements

Consider a domain $\Omega = [0, 2]$ divided into the three elements $[0, 1]$, $[1, 1.2]$, and $[1.2, 2]$.

For P1 and P2 elements, set up the list of coordinates and nodes (`nodes`) and the numbers of the nodes that belong to each element (`elements`) in two cases: 1) nodes and elements numbered from left to right, and 2) nodes and elements numbered from right to left. Filename: `fe_numberings1.py`..

Exercise 11: Define vertices, cells, and dof maps

Repeat Exercise 10, but define the data structures `vertices`, `cells`, and `dof_map` instead of `nodes` and `elements`. Filename: `fe_numberings2.py`.

Exercise 12: Construct matrix sparsity patterns

Exercise 10 describes a element mesh with a total of five elements, but with two different element and node orderings. For each of the two orderings, make a 5×5 matrix and fill in the entries that will be nonzero.

Hint. A matrix entry (i,j) is nonzero if i and j are nodes in the same element. Filename: `fe_sparsity_pattern.pdf`.

Exercise 13: Perform symbolic finite element computations

Perform symbolic calculations to find formulas for the coefficient matrix and right-hand side when approximating $f(x) = \sin(x)$ on $\Omega = [0, \pi]$ by two P1 elements of size $\pi/2$. Solve the system and compare $u(\pi/2)$ with the exact value 1. Filename: `sin_approx_P1.py`.

Exercise 14: Approximate a steep function by P1 and P2 elements

Given

$$f(x) = \tanh(s(x - \frac{1}{2}))$$

use the Galerkin or least squares method with finite elements to find an approximate function $u(x)$. Choose $s = 40$ and try $N_e = 4, 8, 16$ P1 elements and $N_e = 2, 4, 8$ P2 elements. Integrate $f\varphi_i$ numerically. Filename: `tanh_fe_P1P2_approx.py`.

Exercise 15: Approximate a steep function by P3 and P4 elements

Solve Exercise 14 using $N_e = 1, 2, 4$ P3 and P4 elements. How will a collocation/interpolation method work in this case with the same number of nodes? Filename: `tanh_fe_P3P4_approx.py`.

Exercise 16: Investigate the approximation error in finite elements

The theory (93) from Section 6.4 predicts that the error in the Pd approximation of a function should behave as h^{d+1}, where h is the length of the element. Use experiments to verify this asymptotic behavior (i.e., for small enough h). Choose three examples: $f(x) = Ae^{-\omega x}$ on $[0, 3/\omega]$, $f(x) = A\sin(\omega x)$ on $\Omega = [0, 2\pi/\omega]$ for constant A and ω, and $f(x) = \sqrt{x}$ on $[0, 1]$.

Hint. Run a series of experiments: (h_i, E_i), $i = 0, \ldots, m$, where E_i is the L^2 norm of the error corresponding to element length h_i. Assume an error model $E = Ch^r$ and compute r from two successive experiments:

$$r_i = \ln(E_{i+1}/E_i)/\ln(h_{i+1}/h_i), \quad i = 0, \ldots, m-1\,.$$

Hopefully, the sequence $r_0, \ldots, r_{m-1}$ converges to the true r, and r_{m-1} can be taken as an approximation to r. Run such experiments for different d for the different $f(x)$ functions.

Filename: `Pd_approx_error.py`.

Exercise 17: Approximate a step function by finite elements

Approximate the step function

$$f(x) = \begin{cases} 1 & 0 \leq x < 1/2, \\ 2 & 1/2 \leq x \geq 1/2 \end{cases}$$

by 2, 4, and 8 P1 and P2 elements. Compare approximations visually.

Hint. This f can also be expressed in terms of the Heaviside function $H(x)$: $f(x) = H(x - 1/2)$. Therefore, f can be defined by

```
f = sp.Heaviside(x - sp.Rational(1,2))
```

making the `approximate` function in the `fe_approx1D.py` module an obvious candidate to solve the problem. However, `sympy` does not handle symbolic integration with this particular integrand, and the `approximate` function faces a problem when converting `f` to a Python function (for plotting) since `Heaviside` is not an available function in `numpy`. It is better to make special-purpose code for this case or perform all calculations by hand.
Filename: `Heaviside_approx_P1P2.py`.

Exercise 18: 2D approximation with orthogonal functions

Assume we have basis functions $\varphi_i(x, y)$ in 2D that are orthogonal such that $(\varphi_i, \varphi_j) = 0$ when $i \neq j$. The function `least_squares` in the file `approx2D.py`[18] will then spend much time on computing off-diagonal terms in the coefficient matrix that we know are zero. To speed up the computations, make a version `least_squares_orth` that utilizes the orthogonality among the basis functions. Apply the function to approximate

$$f(x, y) = x(1 - x)y(1 - y)e^{-x-y}$$

on $\Omega = [0, 1] \times [0, 1]$ via basis functions

$$\varphi_i(x, y) = \sin((p + 1)\pi x) \sin((q + 1)\pi y), \quad i = q(N_x + 1) + p,$$

where $p = 0, \ldots, N_x$ and $q = 0, \ldots, N_y$.

Hint. Get ideas from the function `least_squares_orth` in Section 2.8 and file `approx1D.py`[19].
Filename: `approx2D_least_squares_orth.py`.

[18] `http://tinyurl.com/nm5587k/fem/fe_approx2D.py`
[19] `http://tinyurl.com/nm5587k/fem/fe_approx1D.py`

Exercise 19: Use the Trapezoidal rule and P1 elements

Consider approximation of some $f(x)$ on an interval Ω using the least squares or Galerkin methods with P1 elements. Derive the element matrix and vector using the Trapezoidal rule (101) for calculating integrals on the reference element. Assemble the contributions, assuming a uniform cell partitioning, and show that the resulting linear system has the form $c_i = f(x_i)$ for $i \in \mathcal{I}_s$. Filename: `fe_P1_trapez.pdf`.

Problem 20: Compare P1 elements and interpolation

We shall approximate the function

$$f(x) = 1 + \epsilon \sin(2\pi n x), \quad x \in \Omega = [0, 1],$$

where $n \in \mathbb{Z}$ and $\epsilon \geq 0$.

a) Plot $f(x)$ for $n = 1, 2, 3$ and find the wave length of the function.

b) We want to use N_P elements per wave length. Show that the number of elements is then nN_P.

c) The critical quantity for accuracy is the number of elements per wave length, not the element size in itself. It therefore suffices to study an f with just one wave length in $\Omega = [0, 1]$. Set $\epsilon = 0.5$.

Run the least squares or projection/Galerkin method for $N_P = 2, 4, 8, 16, 32$. Compute the error $E = ||u - f||_{L^2}$.

Hint. Use the `fe_approx1D_numint` module to compute u and use the technique from Section 6.4 to compute the norm of the error.

d) Repeat the set of experiments in the above point, but use interpolation/collocation based on the node points to compute $u(x)$ (recall that c_i is now simply $f(x_i)$). Compute the error $E = ||u - f||_{L^2}$. Which method seems to be most accurate?
Filename: `P1_vs_interp.py`.

Exercise 21: Implement 3D computations with global basis functions

Extend the `approx2D.py`[20] code to 3D applying ideas from Section 8.4. Use a 3D generalization of the test problem in Section 8.3 to test the implementation. Filename: `approx3D.py`.

Exercise 22: Use Simpson's rule and P2 elements

Redo Exercise 19, but use P2 elements and Simpson's rule based on sampling the integrands at the nodes in the reference cell. Filename: `fe_P2_simpson.pdf`.

[20] `http://tinyurl.com/nm5587k/fem/approx2D.py`

References

[1] M. G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer, 2013.

Index